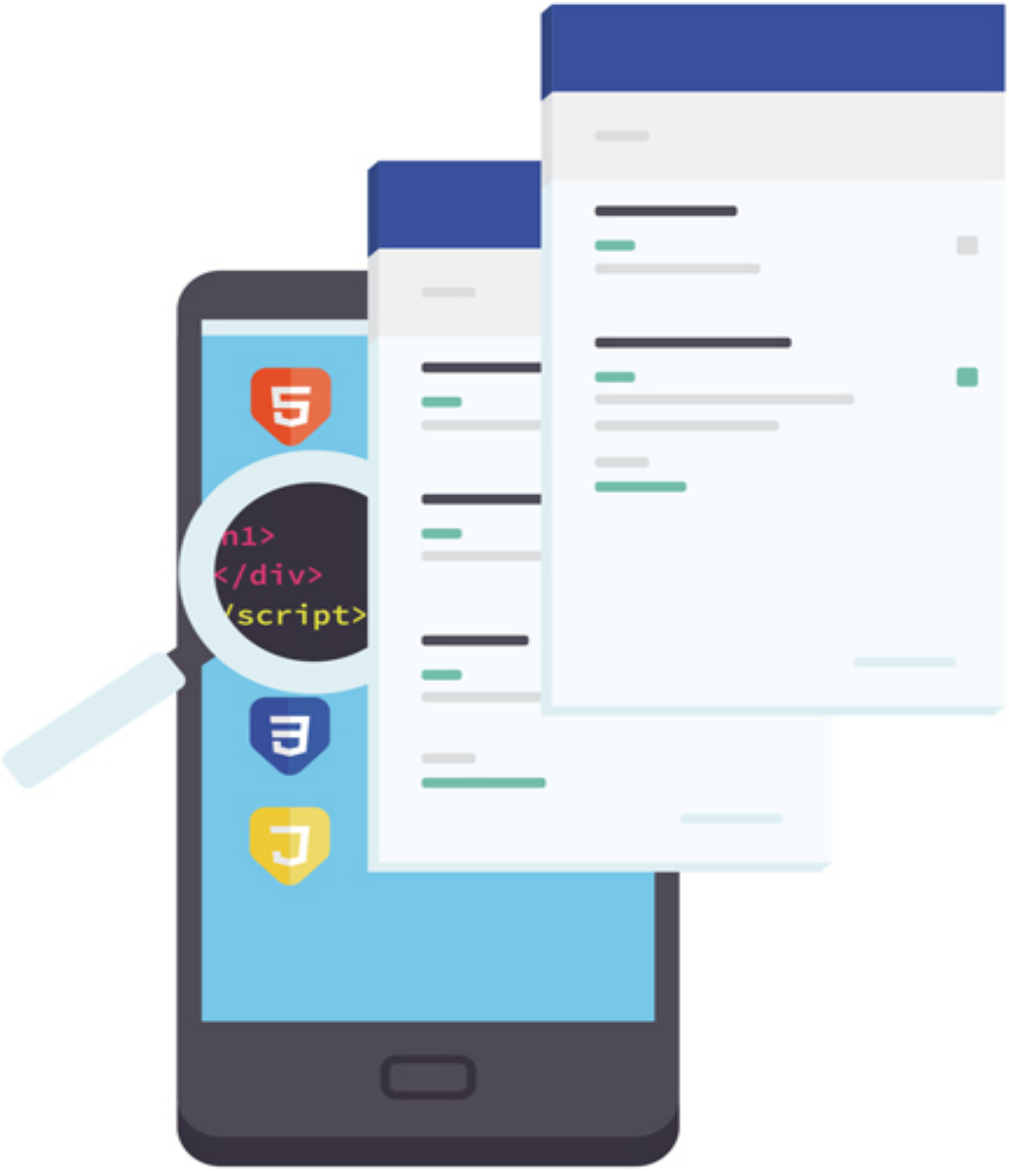
Cordova avançado e PhoneGap

Um guia detalhado do zero à publicação



Casa do Código

DIOGO MACHADO

Sumário

- 1. Introdução ao PhoneGap e ao Cordova
- 2. A primeira versão do aplicativo Freeburguer
- 3. O design do aplicativo e a experiência do usuário
- 4. Desenvolvendo as funcionalidades do aplicativo
- 5. Detalhes exclusivos da plataforma Android
- 6. Detalhes exclusivos da plataforma iOS
- 7. Hooks, armazenamento local, segurança e muito mais!
- 8. Continuando os estudos
- 9. Apêndice A – Configurando seu ambiente

Capítulo 1
Introdução ao PhoneGap e ao Cordova

O framework PhoneGap nasceu em 2008, no evento chamado *iPhoneDevCamp* em São Francisco, estado da Califórnia. Foi desenvolvido originalmente por Rob Ellis, Brock Whitten, Brian Leroux, Joe Bowser e Dave Johnson, todos faziam parte do time da empresa *Nitobi Software*, localizada em Vancouver (Canadá).

O projeto surgiu silenciosamente em 2008, mas foi ganhando sua notoriedade principalmente devido à **curva de aprendizagem** da linguagem JavaScript, por ser menor quando comparada a uma linguagem nativa, como o Java (do Android), o Swift ou o Objective-C (do iOS) e o C# (do Windows Phone). O crescimento também se deu pelo fato de o JS ser uma linguagem muito comum entre os desenvolvedores web (top 10 há anos entre as mais populares no índice Tiobe).

> **Índice Tiobe**
>
> O índice é uma lista ordenada de linguagens de programação, classificada pela frequência de pesquisa na web usando o nome da linguagem como palavra-chave: https://www.tiobe.com/tiobe-index/.

De 2008 a 2009, a *Nitobi* foi adicionando recursos, realizando eventos e, aos poucos, expandindo o universo híbrido que estava surgindo. Em 2010, a *Apple* confirmou que a plataforma estava de acordo com as regras da licença para desenvolvedores iOS – mais um grande passo para a expansão do framework e o crescimento dos apps híbridos.

Mais tarde, em 4 de outubro de 2011, a *Adobe* anunciou a aquisição da *Nitobi*. Com isso, a empresa resolveu abrir o código-fonte para a

comunidade, repassando a responsabilidade dele para a *Apache Software Foundation* gerir. Nessa transição, o código-fonte do framework PhoneGap foi renomeado para **Apache Cordova**.

O nome PhoneGap não pode ser repassado porque a marca foi registrada pela *Nitobi*, logo, a *Adobe* resolveu manter o nome PhoneGap para servir comercialmente. Assim, hoje ela distribui um "*Cordova incrementado*", com ferramentas e serviços exclusivos que não fariam sentido algum estarem no Apache Cordova.

> **PhoneGap versus Apache Cordova**
>
> Podemos dizer que o PhoneGap possui todos os recursos que o Cordova, mas o Cordova não apresenta todos os recursos do PhoneGap. Ainda assim, o Cordova é o coração do framework, por isso falaremos muito dele.

Em setembro de 2012, a *Adobe* lançava o PhoneGap Build, um serviço em nuvem para facilitar a geração de aplicativos para diversas plataformas. O objetivo foi tirar dos desenvolvedores a complexidade de se configurar ambientes de desenvolvimento (isso inclui baixar e configurar cada SDK) para cada plataforma.

Ainda em 2012, a Wikipédia lançou uma aplicação produzida totalmente com PhoneGap para iOS, atraindo ainda mais a atenção dos desenvolvedores para o framework. Ainda nesse mesmo ano, surgiram muitas atualizações, e um dos destaques importantes desta época é o suporte adicionado ao *Adobe* Dreamweaver CS6 para a plataforma PhoneGap Build.

Já em 2013, o framework seguiu a todo vapor, com lançamentos frequentes de novos plugins que davam mais poder de acesso ao hardware. Outro recurso que ajudou a alavancar foi o surgimento de frameworks robustos de front-end que trabalham com o modelo SPA (*Single-page Application*). Para a época, os mais robustos eram o Sencha Touch e o AngularJS. Também neste ano, podemos

destacar o suporte à plataforma Firebase (hoje pertencente ao Google).

De 2014 para cá, o PhoneGap e o Apache Cordova evoluíram muito, em termos de plugins, que são os responsáveis por fazer a ligação nativa com o hardware. Para termos uma ideia, existem cerca de 3.084 plugins (dados extraídos do site oficial do Cordova em janeiro de 2018) disponíveis para uso nas diversas plataformas suportadas (Android, iOS, Windows, Blackberry, Ubuntu, FirefoxOS, Fire OS, WP8 e Browser).

Também atrelada à evolução desses últimos anos, está a própria evolução das tecnologias web que hoje temos à disposição. Estamos falando das APIs do HTML5 como Canvas, SVG, Web Storage, File API, Drag and Drop, Web Sockets. Elas proporcionaram o acesso a recursos incríveis direto no navegador, o que tem tudo a ver com o desenvolvimento híbrido de aplicativos, já que trabalhamos em um navegador embutido ao usar o Cordova, o WebView.

Outros projetos também usam a base do Apache Cordova

Com o crescimento da popularidade do framework, nos últimos anos começaram a surgir também outras iniciativas que se beneficiam do código aberto do Apache Cordova para prover recursos e ferramentas exclusivas. Podemos destacar:

- Ionic Framework
- Monaca
- Telerik Platform
- Intel XDK
- TACO

É claro, cada um desses projetos possui sua própria documentação e particularidades, mas, a grosso modo, podemos disser que eles não existiriam se não tivéssemos hoje sua base, o Apache Cordova.

Desmistificando o framework

1. Apesar de terem a mesma base, PhoneGap CLI, Apache Cordova CLI e Ionic CLI possuem diferenças. CLI é a sigla para linha de comando (ou em inglês, *Command Line*). O que acontece é que cada cliente de comando tem como base o uso do Cordova, mas as suas diferenças estão nos serviços extras.

   Por exemplo, no PhoneGap CLI, é possível publicar um aplicativo direto em um servidor na nuvem da Adobe, chamado PhoneGap Build, algo que não está disponível na linha de comando pura do Apache Cordova CLI. A linha de comando do Ionic também tem suas particularidades, pois ela oferece serviços exclusivos e customizados que só existem em sua plataforma. No geral, todos têm os comandos básicos do Apache Cordova CLI, porém o Ionic e o PhoneGap introduzem comandos extras que trazem uma maior versatilidade para as suas plataformas.

2. PhoneGap e Apache Cordova não oferecem interface gráfica. Isso quer dizer que botões, transições, alertas customizados, isso tudo é feito por você com JavaScript e CSS. É possível também usar um framework front-end.

3. Sua aplicação Cordova ou PhoneGap é escrita em JavaScript, pois, na prática, estamos fazendo um sistema web com acesso ao hardware. Esse código será hospedado dentro do aparelho da pessoa, portanto, você não pode usar linguagens de programação back-end (como PHP, ASP, .NET) dentro do seu projeto diretamente, já que lá não teremos um servidor web rodando e, sim, um navegador interpretando código JavaScript – como fazemos em nosso computador.

   Você pode criar APIs (prática mais comum) que vão interagir com sua aplicação via chamadas Ajax, como uma API que entrega uma lista de itens em JSON, ou uma que recebe os

dados enviados do seu aplicativo via formulário HTTP. Este é o cenário que você deve ter em mente.

4. Não é obrigatório usar um framework front-end para construir aplicações com o Cordova/PhoneGap como o Angular, AngularJS, React ou Vue (apesar de serem extremamente úteis e muito poderosos); o JavaScript já faz o trabalho usando os plugins. Entretanto, esses frameworks suprem o que o Cordova e o PhoneGap não têm, que é um sistema de rotas, cache e outras ferramentas. Não podemos esquecer que o código principal do Cordova/PhoneGap serve única e exclusivamente para expor uma API JavaScript que acessa o hardware dos aparelhos por meio de seus plugins.

1.1 Por dentro de um aplicativo híbrido Cordova

Agora que já sabemos como o framework surgiu, vamos conhecer a arquitetura básica de uma aplicação híbrida. Para isso, veja a figura a seguir.

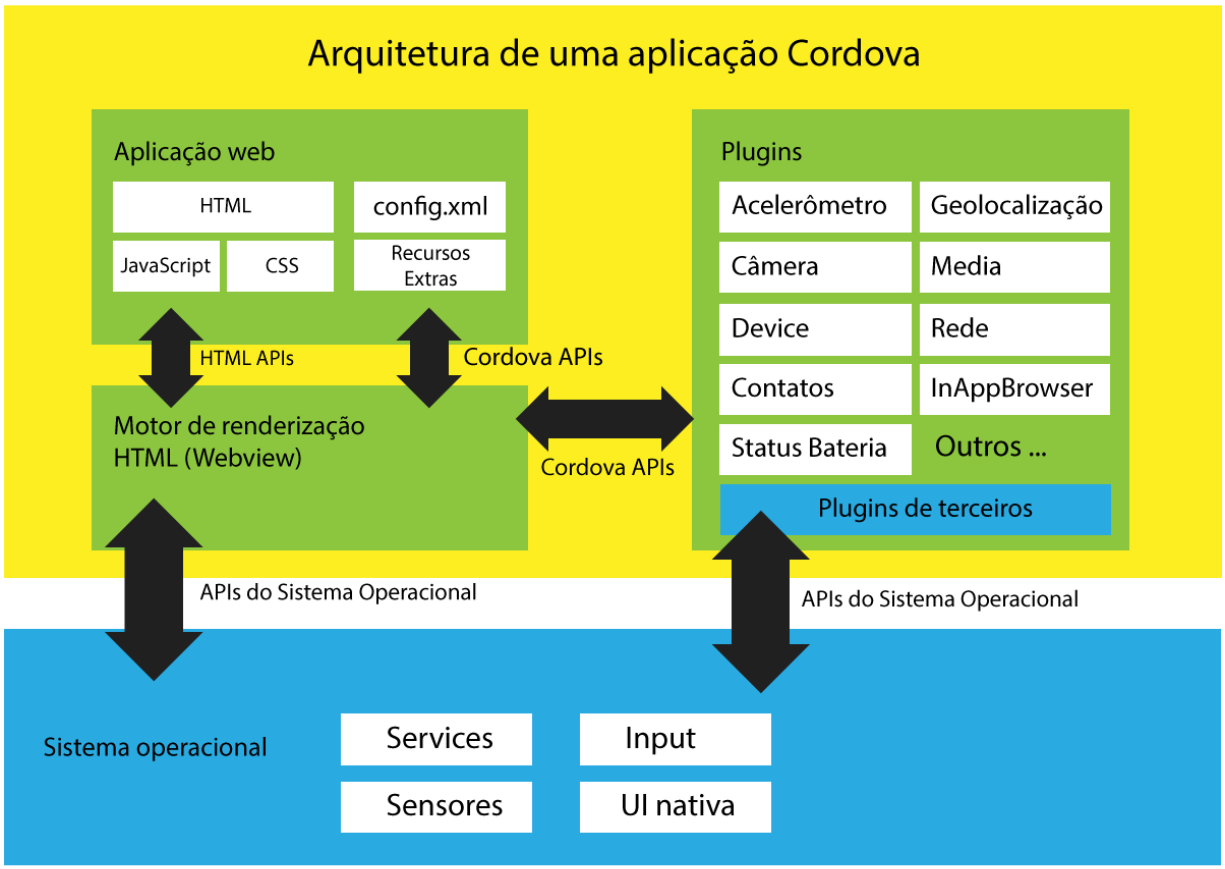


Figura 1.1: Arquitetura de uma aplicação híbrida com Apache Cordova

Sobre a imagem anterior, podemos dizer que:

- **Aplicação web** – É a aplicação escrita em JavaScript, marcada estruturalmente com HTML e embelezada com CSS. O arquivo `config.xml` representa as configurações da aplicação Apache Cordova, sendo os recursos extras outros que precisaremos porventura (como arquivos de áudio, imagens e um arquivo *JSON* etc.)
- **Plugins** – Os plugins são responsáveis pela comunicação com o hardware, e existem cerca de aproximadamente 2.676 prontos para uso.
- **WebView** – É o motor de renderização no qual nossa aplicação será executada; em outras palavras, podemos dizer que é o

navegador de internet onde nossa aplicação roda, invisível aos olhos do usuário, mas importantíssimo no mundo híbrido.
- **Sistema Operacional** – Representam os diversos sistemas operacionais suportados pelo Cordova (Android, iOS, Windows, Blackberry, Ubuntu, FirefoxOS, Fire OS, WP8).

1.2 Objetivo e como ler este livro

No decorrer do livro, vamos construir juntos um projeto completo. No aplicativo, veremos como instalar plugins, integrar a plataforma Firebase, construir elementos visuais do zero usando código web e, o melhor, detalhando todo o processo de publicação. Sempre que necessário, será disponibilizado o código-fonte no GitHub para você baixar e acompanhar.

O principal objetivo é abordar todos os aspectos necessários para você construir bons aplicativos híbridos do zero, preparando-o para enfrentar qualquer desafio.

Sempre que falarmos de *Cordova*, lembre-se de que estamos falando do PhoneGap – a menos que estejamos falando de algo exclusivo do PhoneGap.

> Para saber sobre as configurações do framework no ambiente de desenvolvimento, veja o *Apêndice A – Configurando seu ambiente*.

No próximo capítulo, a *brincadeira* começa a ficar legal. Vamos começar a desenvolver um aplicativo do zero, para conhecer melhor nosso projeto, e a realizar testes em dispositivos reais.

Capítulo 2
A primeira versão do aplicativo Freeburguer

2.1 O projeto Freeburguer

Dois empreendedores nos chamaram para criar o aplicativo **Freeburguer**. Ele realiza pedidos de hambúrgueres e é voltado para casas de lanche e hamburguerias. Cada empresa vai pagar um valor mensal para utilizar a ferramenta e terá um QRCode de identificação; este será utilizado para fazer a divulgação do serviço e também pelos clientes ao abrir o aplicativo e localizar a empresa que deseja fazer o pedido.

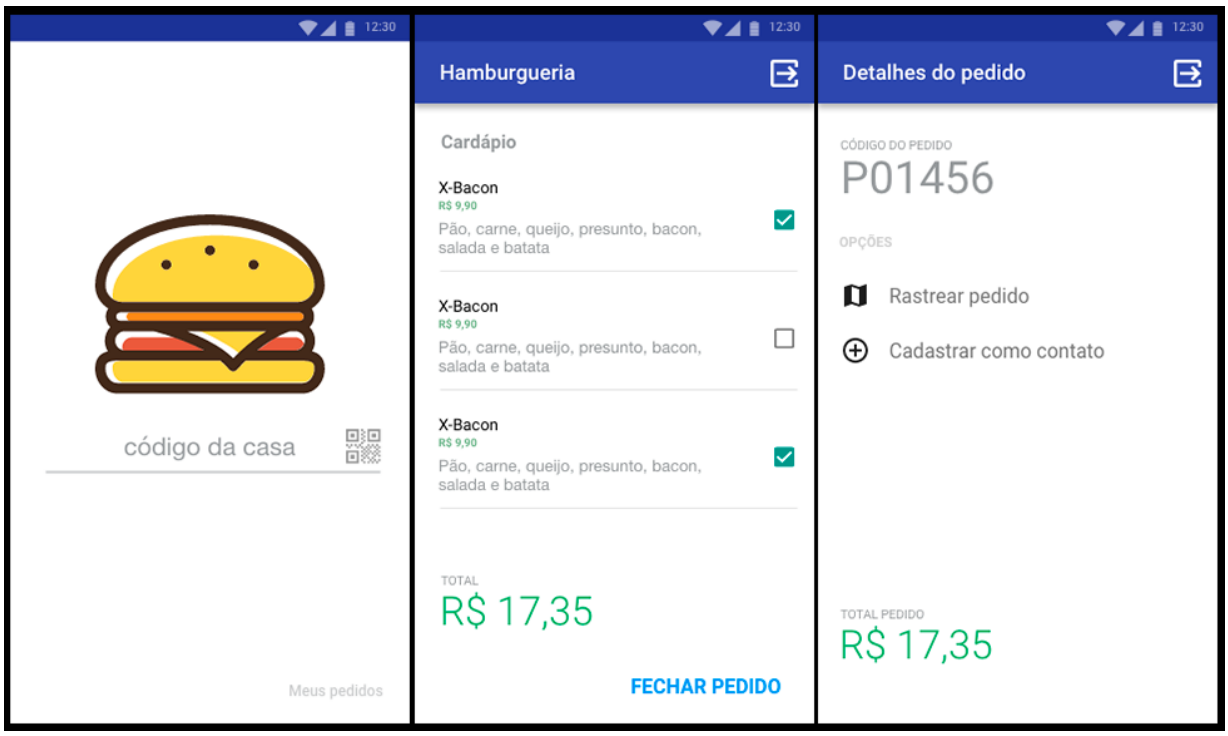


Figura 2.2: Imagem ilustrativa da ideia

Os pedidos serão recebidos em um sistema web, no qual cada empresa-cliente tem sua identificação. Lá elas vão acompanhar os pedidos em tempo real.

Escolhendo um framework SPA

Como já sabemos, o framework Cordova não oferece mais nada além do acesso nativo. Nesse cenário, vamos precisar de uma ajuda para trabalhar a lógica da aplicação: rotas, controladores, visualizações e outras ferramentas. No livro, vamos usar o AngularJS para essa *ajuda*.

Do framework AngularJS, vamos utilizar:

- O seu sistema MVC (*Model, View, Controller*);
- A manipulação da DOM através do Two-Way Data Binding;
- Componentes customizados do HTML via diretivas.

Posso usar React? Vue.js? Angular?

Sim, você é livre para escolher o framework SPA mais adequado para o seu projeto. Neste livro, escolhi trabalhar o AngularJS, pois é de meu conhecimento e ainda é um framework bastante popular.

AngularJS terá suporte LTS por 3 anos ainda

A equipe do Google anunciou uma última versão para o framework AngularJS, a 1.7. Eles ainda divulgaram que, após o lançamento dessa versão, o framework entrará em *Long Term Support* (LTS) por 3 anos, devido à grande quantidade de aplicações e desenvolvedores que ainda usam a versão, dando maior tempo para que todos migrem no futuro para o Angular – a nova versão que mudou drasticamente sua sintaxe apesar de suas melhorias. Saiba mais sobre o anúncio em http://bit.ly/2DPoWpU.

2.2 Criando o projeto do zero com um template

O Cordova e o PhoneGap possuem um parâmetro incrível chamado *template*, que abre a possibilidade de iniciar um projeto novo usando uma estrutura pronta. A fonte do download será o NPM por padrão, mas você também pode usar um caminho de um repositório do GitHub, ou um caminho local na sua máquina.

Seguindo a ideia de se iniciar um projeto Cordova usando template, usaremos um que já esteja ajustado para trabalharmos em conjunto com o AngularJS e o que foi criado por mim. Abra o terminal e execute o comando a seguir:

```
cordova create freeburguer io.apps.freeburguer --template cordova-ng-boilerplate
```

Vale notar que, para adiantar, passamos também um parâmetro logo após o nome `freeburguer` chamado `io.apps.freeburguer`. Esse nome é um **ID único** para a loja de aplicativos (ou seja, não pode existir dois aplicativos com o mesmo ID). Ele está definido assim `id="io.apps.freeburguer"`, dentro do arquivo `config.xml`.

O ID segue o padrão de estilo domínio reverso para empacotamento da aplicação, mas você pode definir do jeito que achar melhor. Um exemplo seria usar o nome da sua empresa, ou sobrenome mais o nome da aplicação. Escolha sempre essa nomenclatura com cuidado, porque não vamos poder mudá-la no futuro depois de lançar o aplicativo.

Apesar de podermos usar um *template* com o comando do Cordova, existem diretórios e arquivos básicos que são criados em uma aplicação Cordova. A seguir, você confere esses arquivos na imagem. Falaremos em um momento oportuno de cada um deles.

```
-rw-r--r--  1 diogomachado  staff   992B 22 Jan 21:28 config.xml
drwxr-xr-x  3 diogomachado  staff   102B 22 Jan 21:28 hooks
-rw-r--r--  1 diogomachado  staff   361B 22 Jan 21:28 package.json
drwxr-xr-x  2 diogomachado  staff    68B 22 Jan 21:28 platforms
drwxr-xr-x  2 diogomachado  staff    68B 22 Jan 21:28 plugins
drwxr-xr-x  5 diogomachado  staff   170B 22 Jan 21:28 res
drwxr-xr-x_ 6 diogomachado  staff   204B 22 Jan 21:28 www
```

Figura 2.3: Estrutura básica do Cordova

O arquivo `config.xml` é a base de toda a nossa aplicação, pois é nele que dizemos o nome da aplicação, quais plugins estão inseridos nela, as configurações de segurança e outras permissões. Quando criamos um projeto vazio, o arquivo vem bem simples, mas, de acordo com o desenrolar do desenvolvimento, este vai obviamente ganhando mais configurações.

2.3 Detalhes do config.xml que ninguém lhe contará

Este é o arquivo que dita as regras da nossa aplicação. Alguns detalhes e dicas importantes que aprendi com o tempo não estão bem explicados na documentação do Cordova e vou passá-los para você aqui.

Tag name

Este é o nome da aplicação. É bom evitar nomes muito grandes para essa tag – no máximo, duas palavras e, se for possível, prefira um sem acentuação. Em alguns momentos, já passei por problemas de quebra do *build* relacionados à acentuação brasileira no `name`. Isso não é uma regra, mas é bom evitar.

Atenção especial à tag widget

Devemos ficar atentos aos parâmetros `id` e `version`. O `id` é aquele que definimos logo ao criar o Freeburguer e chama-se `io.apps.freeburguer`. Essa identificação é única para a loja de aplicativos e **não pode** ser alterada depois que você faz o primeiro envio para a loja.

O `version` dita a versão em que sua aplicação está. Existe também parâmetros alternativos de versão, específicos para plataforma em que vamos trabalhar. Voltaremos a falar disso no capítulo específico de cada plataforma.

Página principal

O Cordova entende que a página principal da nossa aplicação é o arquivo `index.html`, localizado na raiz do diretório da aplicação `freeburguer -> www`. Essa configuração está bem definida na tag `content`, como mostra o código seguir:

```
<content src="index.html" />
```

É importante tomar cuidado, porque, dentro de um projeto Cordova, só pode existir um arquivo `index.html`. Às vezes, ao usar um gerenciador de componentes (como o *Bower*), ou mesmo copiar um diretório para dentro do `www`, você pode acabar levando outro arquivo `index.html` junto. Muitos projetos, principalmente dependências baixadas via *Bower*, possuem um diretório chamado `test`, e geralmente este tem um `index.html`. Isso **quebrará a aplicação**. Nestes casos, o correto é excluir esses arquivos para termos apenas um `index.html`.

Adição de plugins e seu versionamento

Ao longo do desenvolvimento do aplicativo, vamos adicionar plugins que darão acesso a funcionalidades nativas. Uma característica comum de aplicativos nativos é a sua integração com o visual da plataforma que está rodando.

Em um projeto de aplicativo, um ponto principal é ter uma boa *Status Bar* (barra de notificações) configurada. Este é um ponto importante da interface a ser considerado ao configurarmos uma aplicação nova, mas acontece que o Cordova sozinho não trata automaticamente os detalhes de cada plataforma. Por isso, o plugin da Status Bar será o primeiro que vamos adicionar ao Freeburguer.

No diretório do projeto e com o terminal aberto, execute o comando a seguir:

```
cordova plugin add cordova-plugin-statusbar --save
```

O parâmetro `--save` grava no arquivo `config.xml` que o Freeburguer utiliza esse plugin, e isso é importante para que o Cordova baixe todas as dependências automaticamente, caso não as tenha. Outra vantagem é que se todos os seus plugins estão definidos no `config.xml`, você não precisa rastrear o diretório de plugins com a ferramenta Git.

Até aqui, o nosso arquivo `config.xml` encontra-se com:

- Configuração básica de nome e descrição, e página principal configurada em `<content>` como `index.html`.
- Plugin `cordova-plugin-whitelist`, que, por padrão, já vem adicionado ao executar o comando de `create` do Cordova. No *capítulo Hooks, armazenamento local, segurança e muito mais!*, explico em detalhes sua utilidade.
- Plugin `cordova-plugin-statusbar` para tratar a barra de notificações nativa de forma inteligente, fazendo nosso app parecer mais nativo.

Permissão especial no iOS

Alguns plugins no iOS exigem a definição de uma tag de permissão chamada `feature`, definida dentro do `config.xml`. Para o `StatusBar` rodar direito no iOS, precisamos do código a seguir:

```
<feature name="StatusBar">
    <param name="ios-package" value="CDVStatusBar" />
    <param name="onload" value="true" />
</feature>
```

Entendendo o spec e o versionamento semântico

Quando executamos o comando para adicionar o plugin da `StatusBar`, o Cordova sozinho definiu uma linha no arquivo `config.xml`, como a seguir:

```
<plugin name="cordova-plugin-statusbar" spec="~2.1.3" />
```

Preste bastante atenção agora, pois o Cordova não documenta isso muito bem, e quero que você entenda a lógica por trás do parâmetro `spec`. Acontece que o uso do acento til (`~`) no `spec` faz referência ao suporte que o Cordova dá ao *fuzzy version* (versão difusa). Isso está ligado diretamente a uma metodologia de versionamento, chamada de **versionamento semântico**. Nesse padrão, a identificação de uma nova versão segue uma estrutura como a seguir:

Figura 2.4: Sistema de versionamento semântico

Podemos ler agora a versão de um plugin, sabendo que:

- **MAJOR** – É quando ocorrem grandes mudanças que alteram a API anterior;
- **MINOR** – Quando ocorrem adição de funcionalidades, mas a compatibilidade é mantida;
- **PATCH** – Ocorrem correções de falhas e a compatibilidade é mantida.

Entender esse funcionamento das versões nos garante saber escolher um plugin e sua versão, e, ainda mais, saberemos que o código que estamos desenvolvendo será compatível com a versão do plugin que estamos usando, porque ele foi bem definido. Agora entendemos que o acento til é usado para fazer o *fuzzy version*, e verificar se uma nova versão foi lançada.

Para concluirmos a ideia do versionamento, podemos imaginar que, quando rodarmos um comando como o `cordova platform add android`, ele vai verificar o arquivo `config.xml` e atenderá aos plugins baseado no `spec`. Logo, para o nosso caso da `StatusBar`, se um novo `PATCH` for lançado, o Cordova vai atualizar o diretório do plugin automaticamente, sem que seja necessário mudarmos manualmente o `spec` do plugin no `config.xml`.

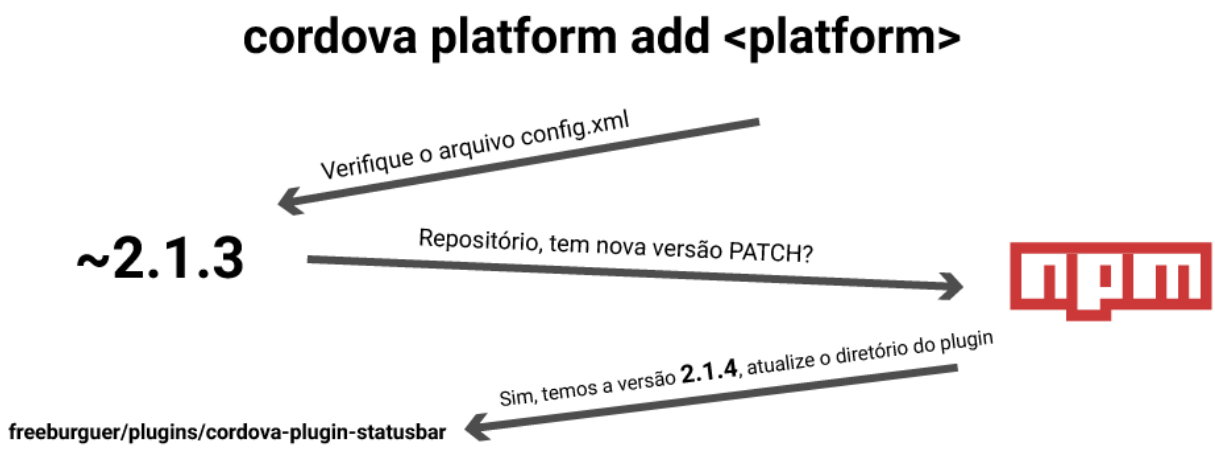


Figura 2.5: Explicação de como ocorre uma atualização

Na maioria dos casos – e no nosso também –, já é suficiente o uso do til para fazer o *fuzzy version* que o próprio Cordova já configura quando rodamos o comando de adicionar o plugin.

OUTRAS OPÇÕES DE VERSIONAMENTO NO SPEC

Existem outras opções que podem ser utilizadas ao adicionar e configurar a versão de um plugin. Uma delas é adicionar um plugin em uma versão específica, bastando usar uma *arroba* (`@`):

```
cordova plugin add cordova-plugin-statusbar@2.1.3 --save
```

Ou definir direto no `config.xml` :

```
<plugin name="cordova-plugin-splashscreen" spec="2.1.3" />
```

A outra opção seria definir o `spec` sem MINOR e PATCH. Assim, o plugin estaria sempre buscando as versões com funcionalidades mais atualizadas daquela versão MAJOR:

```
<plugin name="cordova-plugin-splashscreen" spec="~2" />
```

A última opção é a de buscar o mais recente usando *asterisco*:

```
<plugin name="cordova-plugin-splashscreen" spec="*" />
```

2.4 Configurando uma boa StatusBar

Android e iOS possuem uma pequena diferença quando se trata de cores na StatusBar. Essa diferença será corrigida com o plugin que acabamos de adicionar ao projeto, o da StatusBar.

- **O padrão Android** – No Android, queremos que a StatusBar tenha uma cor um pouco mais escura do que o cabeçalho principal da aplicação, seguindo a ideia do Material Design, por isso escolhemos a cor `#283593` .
- **O padrão iOS** – No iOS, o cabeçalho e StatusBar são uma cor única, portanto, seguimos a cor principal, `#3949AB` . Se não

fizéssemos a configuração do plugin com a cor principal, teríamos uma barra preta por padrão.

Dentro do arquivo `config.xml`, tratamos essas diferenças:

```
<platform name="android">
    <preference name="StatusBarBackgroundColor" value="#283593" />
</platform>

<platform name="ios">
    <preference name="StatusBarOverlaysWebView" value="false"/>
    <preference name="StatusBarBackgroundColor" value="#3949AB"/>
</platform>
```

A preferência `StatusBarOverlaysWebView` setada como `false` garante que o cabeçalho com o título da aplicação não seja sobreposto na `Webview`. Assim, o título *Freeburguer* não ficará embaixo dos ícones principais do iOS.

> **DICA**
>
> Sempre que tiver interesse em ver mais configurações de um plugin, você pode buscar por ele em seu repositório oficial. No caso do Cordova, por padrão, os plugins são publicados no site: https://www.npmjs.com.
>
> O link direto para o plugin do StatusBar é https://www.npmjs.com/package/cordova-plugin-statusbar.

Agora, apenas para efeito de **teste**, crie um cabeçalho que exibirá o título da aplicação dentro da aplicação, em `www -> app -> views -> home.html`. A classe `.header` vai receber um background `#3949AB`, e terá a marcação em HTML como o código a seguir:

```
<header class="header">
  <h1 class="header-title">Freeburguer</h1>
</header>
```

No arquivo `app.css`, localizado em `www -> assets -> css`, temos a definição do fundo:

```
.header{
    background: #3949AB;
    padding: 4.5%;
}

.header-title{
    margin: 0;
    color: white;
    font-size: 1.6em;
    font-family: Arial;
    font-weight: normal;
}
```

Com essas configurações, conseguimos uma experiência mais nativa para cada plataforma – o que é ótimo, pois o usuário já está acostumado.

2.5 Configurando splashscreens e ícones no Android e iOS

Decidimos usar uma tela de abertura, também conhecida como splashscreen, para dar um efeito visual mais interessante para o nosso aplicativo. Porém, vale lembrar que nem todo aplicativo precisa usar esse recurso.

Um outro detalhe é que aqui vamos utilizar o splashscreen nativo do sistema por meio de um plugin, mas ele poderia facilmente ser *simulado* usando HTML, JavaScript e CSS. Não existe melhor opção, e sim aquela que se adequa melhor ao seu projeto.

Uma das desvantagens de se ter um splashscreen é que você prolonga a entrada no app, pois é necessário configurar o tempo de exibição da imagem. Por isso, alguns aplicativos (como o *discador*

do telefone) não o possuem, já que se tornaria muito cansativo termos de ver uma imagem com dois ou três segundos de exibição a cada ligação.

É claro que existem customizações que você pode fazer com o plugin. Para mais detalhes, acesse a documentação em http://bit.ly/2FhTwpr.

Na documentação do Cordova, tem tudo explicado sobre a configuração dos tamanhos dos ícones e splashscreens, porém esse é um trabalho manual bem cansativo. Para nos ajudar, podemos usar um recurso da linha de comando do Ionic para gerar os splashscreens e ícones do Android e iOS, como expliquei no início do livro.

Cordova, PhoneGap e Ionic possuem ferramentas diferentes na linha de comando, em especial no Ionic e no PhoneGap, que tentam ofertar ferramentas extras para desenvolvedores que trabalham com híbrido. No nosso caso, vamos usar a linha de comando do Ionic para nos ajudar com a geração dos *assets*. Você não precisa conhecer previamente a ferramenta.

Como decidimos usar splashscreen, primeiro precisamos adicionar o plugin ao projeto. Sendo assim, dentro do diretório do Freeburguer com o terminal aberto, rodamos o seguinte comando:

```
cordova plugin add cordova-plugin-splashscreen --save
```

O QUE ACONTECE QUANDO ADICIONAMOS UM PLUGIN?

Ao adicionar um plugin com o comando do Cordova, seu código será adicionado ao diretório `plugins`. Esse diretório conterá o código necessário para cada plataforma que ele suporta, como Android, iOS, Windows e assim por diante. Esse código será usado na hora de gerar o aplicativo.

O próximo passo é gerar os arquivos de splashscreen e os ícones para atender aos diversos tamanhos de telas. Este passo poderia ser feito manualmente, mas vamos usar o cliente de linha de comando do Ionic apenas para fazer essa tarefa e nos poupar tempo. Ele também é distribuído pelo NPM, logo, podemos instalá-lo no terminal:

```
npm install -g ionic
```

Depois da instalação do Ionic, precisamos preparar dois arquivos que servirão de base para o comando `ionic resources` gerar os splashscreens e os ícones (do Android e do iOS).

Para usar o `ionic resources`, é necessário criar um diretório chamado `resources` no caminho `Freeburguer -> resources`. Nesse diretório, vamos armazenar dois arquivos base: um chamado `splash` e outro `icon`. Claro que também informaremos a extensão, exemplo: `splash.png` e `icon.png`.

Esses arquivos são criados por nós no *Photoshop* ou *Illustrator*, por exemplo. Eles servem de base para gerar os vários tamanhos de ícones e splashscreens nas plataformas do Android e do iOS. Os formatos aceitos pelo comando Ionic são `.ai`, `.png` e `.psd`.

> Você pode baixar os arquivos `icon.png` e `splash.png` do Freeburguer aqui: http://bit.ly/2b6nA8m.

A dimensão recomendada do arquivo `icon` é de, no mínimo, 1024x1024. E é recomendado que a imagem do arquivo `splash` tenha dimensão 2208x2208. A arte deve estar centralizada em um espaço de 1200x1200, para que se tenha uma área de corte; caso contrário, ela pode não se enquadrar bem em todos os splashscreens gerados.

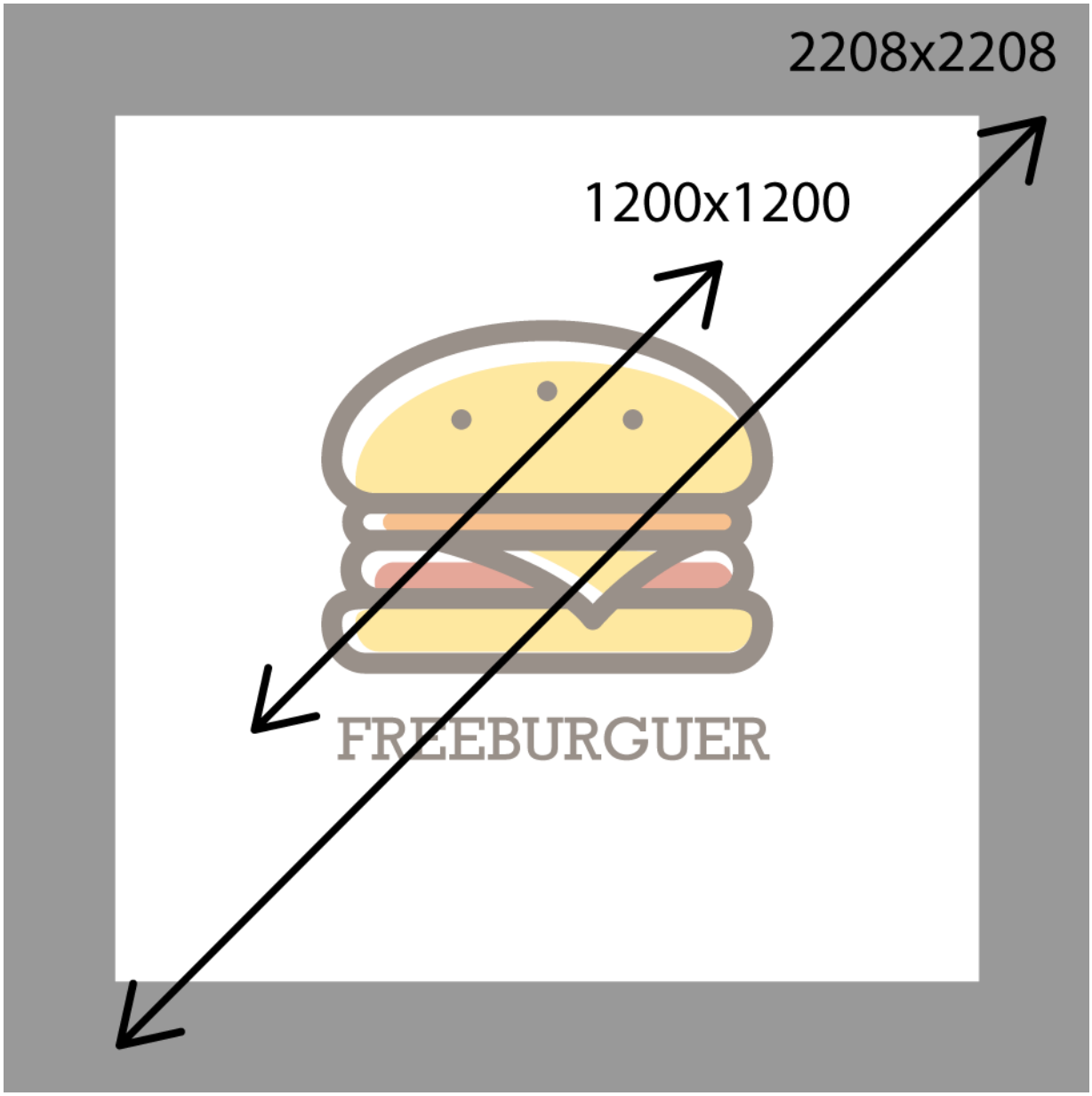


Figura 2.6: A área cinza representa o espaço deixado para a ferramenta cortar a imagem

Como o projeto Freeburguer não foi criado usando o comando do Ionic – mas, mesmo assim, vamos usar um comando Ionic –, vamos precisar de um arquivo chamado `ionic.config.json` dentro do diretório da nossa aplicação. O problema é que esse arquivo só existe quando criamos um novo projeto com Ionic. A dica aqui é criar um projeto novo separado, usando `ionic start`, e depois copiar o arquivo para dentro da nossa aplicação.

Fora do diretório do Freeburguer, com terminal aberto, rodamos o comando a seguir para criar um projeto novo com Ionic:

```
ionic start teste
```

Agora é só entrar no diretório do projeto Ionic chamado `teste`, e copiar o arquivo `ionic.config.json` para dentro da raiz da nossa aplicação. Vale lembrar de que fizemos isso porque queremos usar o cliente de linha de comando do Ionic para gerar os ícones e splashscreens do nosso aplicativo, para ganhar tempo.

O arquivo `ionic.config.json` tem o seguinte conteúdo – lembrando de que não precisamos nos preocupar com ele:

```
{
  "name": "hello",
  "app_id": "",
  "type": "custom",
  "integrations": {
    "cordova": {}
  }
}
```

Gerando ícones e splashscreens para Android e iOS

Configuramos o básico para usar a linha de comando do Ionic e nos ajudar com o trabalho de gerar ícones e splashscreens de forma automatizada, poupando nosso tempo. Vamos começar com a plataforma Android. Com o terminal aberto, primeiro adicione a plataforma Android ao projeto:

```
cordova platform add android
```

Em seguida, execute o comando `ionic`:

```
ionic cordova resources android
```

Depois de rodar o comando para cada plataforma, será criado um diretório com os arquivos de ícones e splashscreens corretos de cada plataforma.

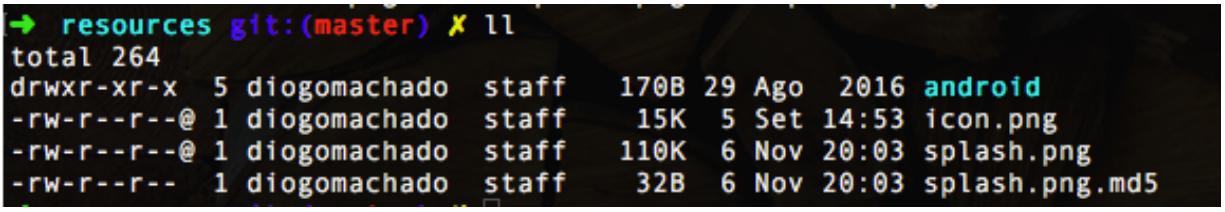

resources git:(master) ✗ ll
total 264
drwxr-xr-x 5 diogomachado staff   170B 29 Ago 2016 android
-rw-r--r--@ 1 diogomachado staff   15K 5 Set 14:53 icon.png
-rw-r--r--@ 1 diogomachado staff   110K 6 Nov 20:03 splash.png
-rw-r--r-- 1 diogomachado staff   32B 6 Nov 20:03 splash.png.md5

Figura 2.7: Arquivos dentro do diretório resources

Repare que, dentro do diretório `resources`, agora temos a pasta `android`, que armazena os ícones e os splashscreens gerados. Para nossa felicidade, o arquivo `config.xml` também foi configurado automaticamente.

Agora, no iOS, vamos usar um arquivo diferente de ícone, porque não vamos inserir um arquivo com fundo transparente nele, e, sim, com fundo branco e sem arredondamento nas laterais. A plataforma iOS vai arredondá-las automaticamente. Faremos essa geração com um arquivo diferenciado para o ícone ficar melhor encaixado no sistema também.

No link http://bit.ly/2b6nA8m, baixe o arquivo `icon-ios.png`, mova-o para um novo diretório `freeburguer -> resources -> ios`, e renomeie-o para `icon.png`, para que o comando possa reconhecê-lo como um arquivo fonte de ícone. Em seguida, execute o comando para gerar os arquivos para a plataforma:

```
ionic cordova resources ios
```

Retornando ao arquivo `config.xml`, percebemos que as seguintes preferências foram definidas automaticamente após o comando:

```
<preference name="SplashScreen" value="screen"/>
<preference name="SplashScreenDelay" value="3000"/>
```

A primeira preferência `SplashScreen` é referente a como o Android armazena o nome das imagens de splashscreen nas pastas `drawable` – normalmente armazenadas em `freeburguer -> platform -> android -> res`:

- platforms/android/res/drawable-hdpi/screen.png
- platforms/android/res/drawable-ldpi/screen.png

A segunda preferência `SplashScreenDelay` é referente ao tempo que o splashscreen ficará visível na tela – por padrão, 3 segundos, o que é bem rápido. Vamos aumentar para 5 segundos. Lembre-se de que o tempo na configuração é marcado em **milissegundos**, por isso, 5 segundos é igual a 5.000 milissegundos:

```
<preference name="SplashScreenDelay" value="5000"/>
```

No Android, temos ainda um parâmetro chamado `FadeSplashScreenDuration`, que aplica um efeito *fade* ao splashscreen. Por padrão, a transição ocorre em 3 segundos, mas vamos mudar para 1. Vale lembrar que temos um *delay* de 5 segundos mais uma transição de 1. Acontece que não somamos esses números, portanto, nossa abertura de splashscreen ainda ocorre em 5 segundos, com o *fade* sendo acionado no quarto segundo.

```
<preference name="FadeSplashScreenDuration" value="1000"/>
```

Outro comportamento do Android é o *Spinner* de carregamento. Ele é verde por padrão, e podemos desativá-lo setando-o como `false`:

```
 <preference name="ShowSplashScreenSpinner" value="false"/>
```

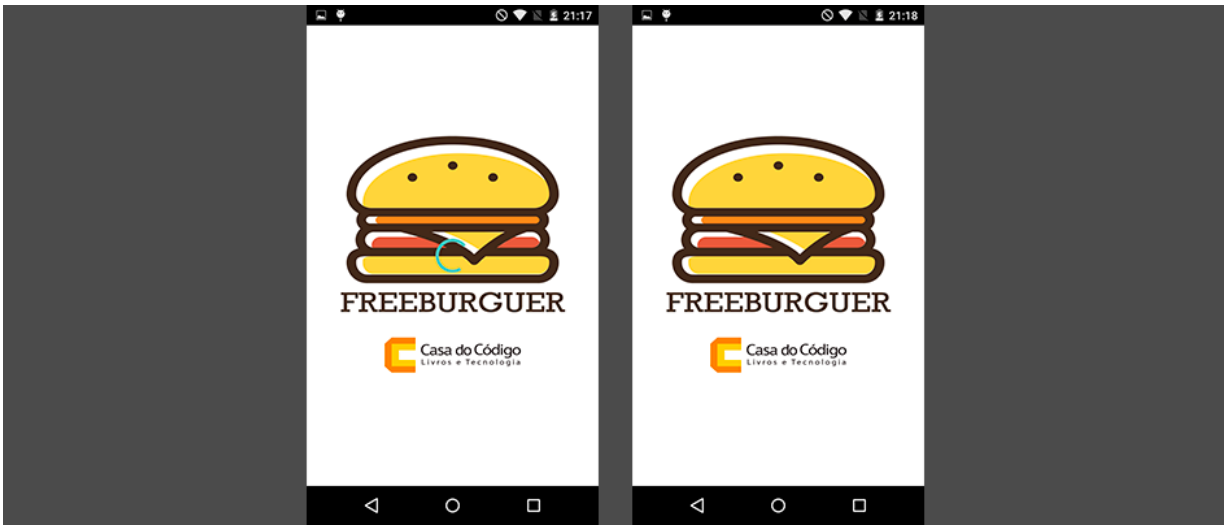


Figura 2.8: Android iniciando com e sem spinner

> Outras preferências do plugin de splashscreen podem ser visualizadas em http://bit.ly/splash-cordova.

2.6 O primeiro teste em um dispositivo real Android

Ative o modo desenvolvedor no aparelho

No menu do aparelho, vá em `Configurar -> Sobre o telefone`, e pressione cinco vezes o `Número da versão`. Agora volte e veja a aba `Programador` disponível. Dentro dessa aba, ative a opção de programador e também a opção de `Depuração USB`, como na figura a seguir:

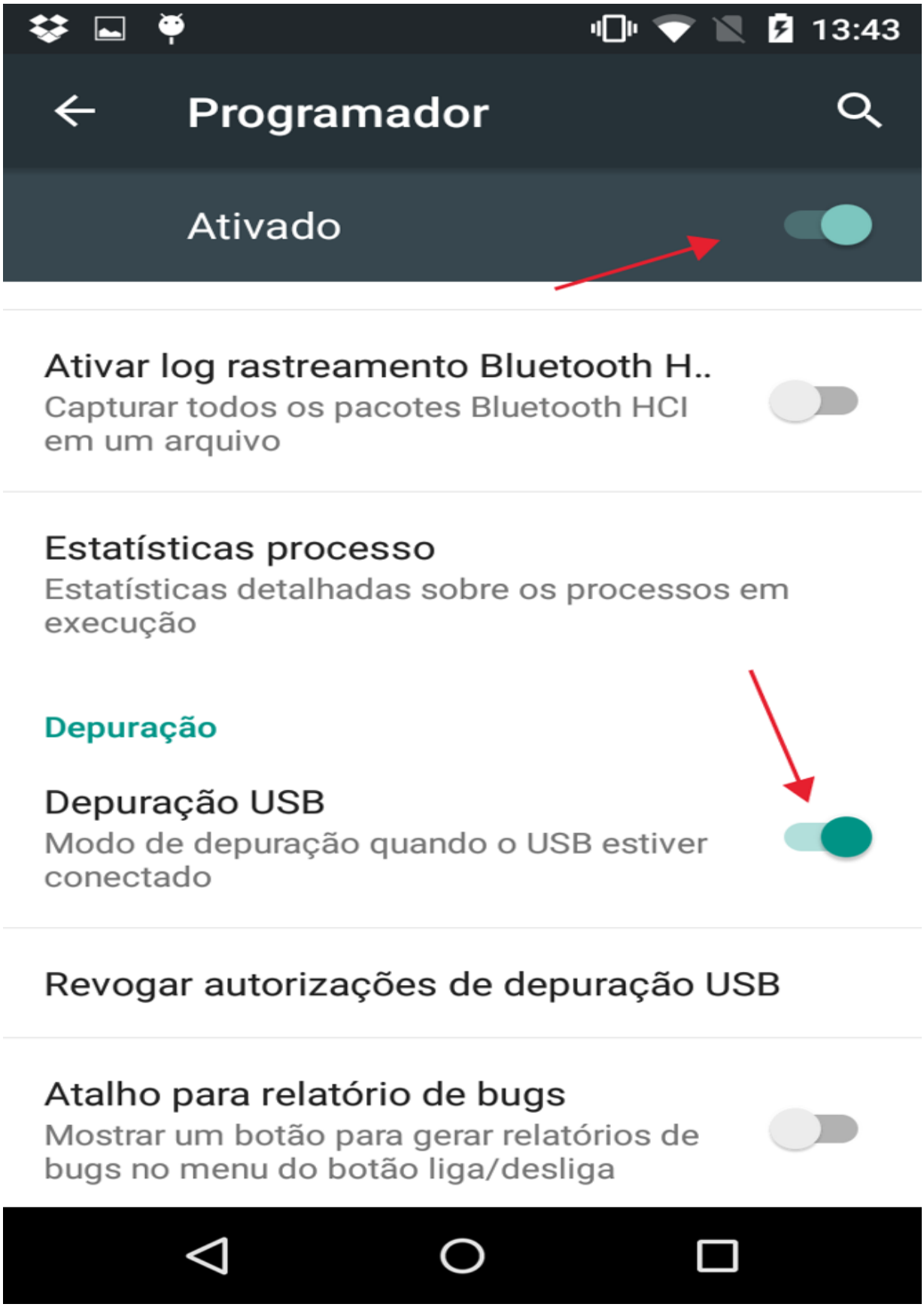

13:43
Programador
Ativado
Ativar log rastreamento Bluetooth H..
Capturar todos os pacotes Bluetooth HCI em um arquivo
Estatísticas processo
Estatísticas detalhadas sobre os processos em execução
Depuração
Depuração USB
Modo de depuração quando o USB estiver conectado
Revogar autorizações de depuração USB
Atalho para relatório de bugs
Mostrar um botão para gerar relatórios de bugs no menu do botão liga/desliga

Figura 2.9: Modo desenvolvedor e depuração USB ativado

> A ativação do *modo desenvolvedor* pode ser diferente em algumas versões do sistema Android, mas a lógica de desbloqueio é a mesma.

Feito a liberação do seu aparelho, conecte-se via USB na sua máquina. Agora, você pode verificar pelo parâmetro `list` quais são os dispositivos disponíveis para executar nossa aplicação:

```
cordova run android --list
```

```
diogomachado@dmachado-desktop:/var/www/livro/freeburguer$ cordova run android --list
Available android devices:
0422213466
Available android virtual devices:
AVD_for_Nexus_One_by_Google
AVD_for_Nexus_One_by_Googlex86
nexus-x86
diogomachado@dmachado-desktop:/var/www/livro/freeburguer$
```

Repare que o ID do nosso aparelho é `0422213466` . Basta rodarmos o comando de execução:

```
cordova run android --device
```

Se você estiver com mais de um aparelho conectado à sua máquina, você tem a opção de rodar passando o ID único:

```
cordova run android --target=0422213466
```

Esse comando vai gerar o `apk` , instalar e executar o aplicativo no seu dispositivo Android.

2.7 O primeiro teste em um dispositivo real iOS

Uma das coisas que mais decepciona o desenvolvedor brasileiro quando começa a desenvolver para iOS é o fato de precisar pagar

99 dólares para ter uma *Apple Developer Account* e poder testar seus aplicativos.

A boa notícia é que hoje já é possível testar um aplicativo direto no iPhone, sem pagar os 99 dólares à Apple. Mas vá se programando, pois, no futuro, para distribuí-lo, você ainda terá de pagar por uma conta de desenvolvedor.

Como explicado no apêndice *Configurando seu ambiente de desenvolvimento*, precisamos do Xcode e inevitavelmente de um computador com MacOS para testar nossa aplicação direto no iPhone. O primeiro passo é adicionar a plataforma `ios` ao projeto:

```
cordova platform add ios
```

Agora, é importante executar o comando `prepare`, porque ele copiará corretamente as configurações do `config.xml` para dentro da plataforma do iOS:

```
cordova prepare
```

Dentro do diretório `freeburguer -> platforms -> ios`, estará disponível um arquivo do tipo *projeto* do Xcode – no nosso caso, chamado `Freeburguer.xcodeproj`. Clique nele duas vezes para abri-lo.

Conecte o iPhone ao computador via USB para que o Xcode identifique o dispositivo que vamos usar para o teste.

Figura 2.11: Seleção de dispositivo no projeto Xcode

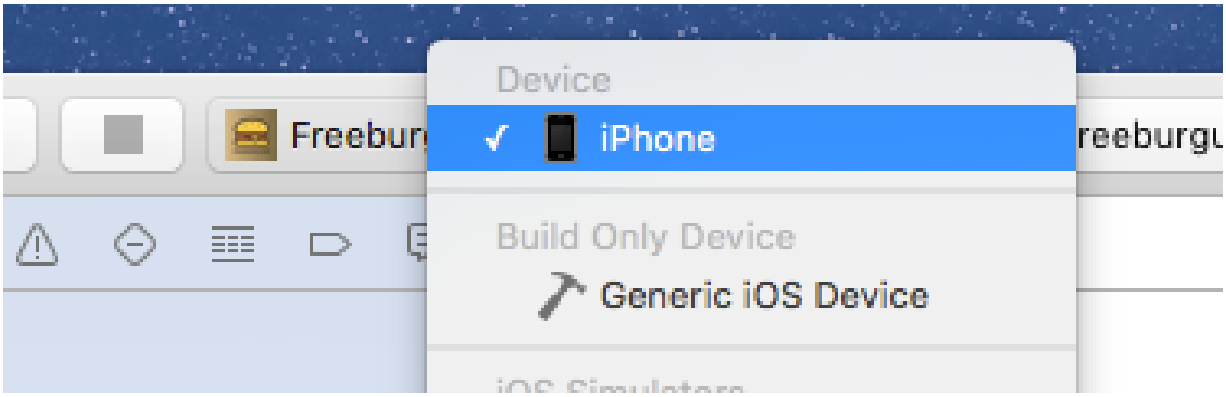


Figura 2.12: Seleção de dispositivo no projeto Xcode

Mesmo que não paguemos nada à Apple neste momento, ainda é necessário usar um *Apple ID*, que qualquer um pode criar de graça no endereço https://appleid.apple.com. No nosso caso, criamos uma conta chamada *apple@diogomachado.com*. Abra o Xcode, no menu superior do sistema, e selecione `XCode -> Preferences`. Na aba `Accounts`, vamos adicionar uma nova conta Apple ID, aquela que acabamos de criar.

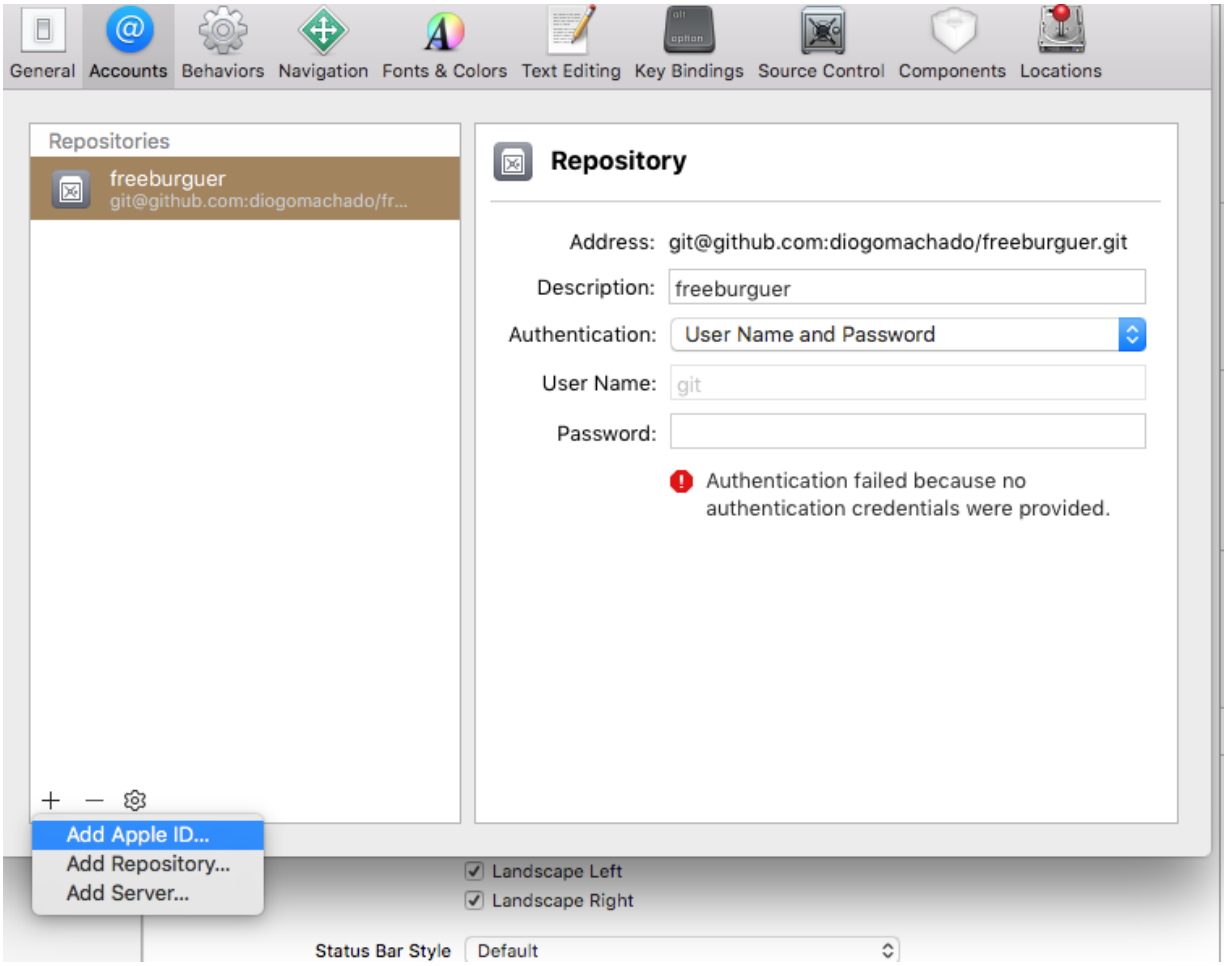


Figura 2.13: Adicionando uma nova conta no Xcode

Com a conta adicionada ao Xcode, podemos executar o projeto no dispositivo. A princípio, teremos um problema já conhecido de assinatura. Este será corrigido ao selecionarmos o `Personal Team` que acabamos de adicionar, aquele nosso Apple ID.

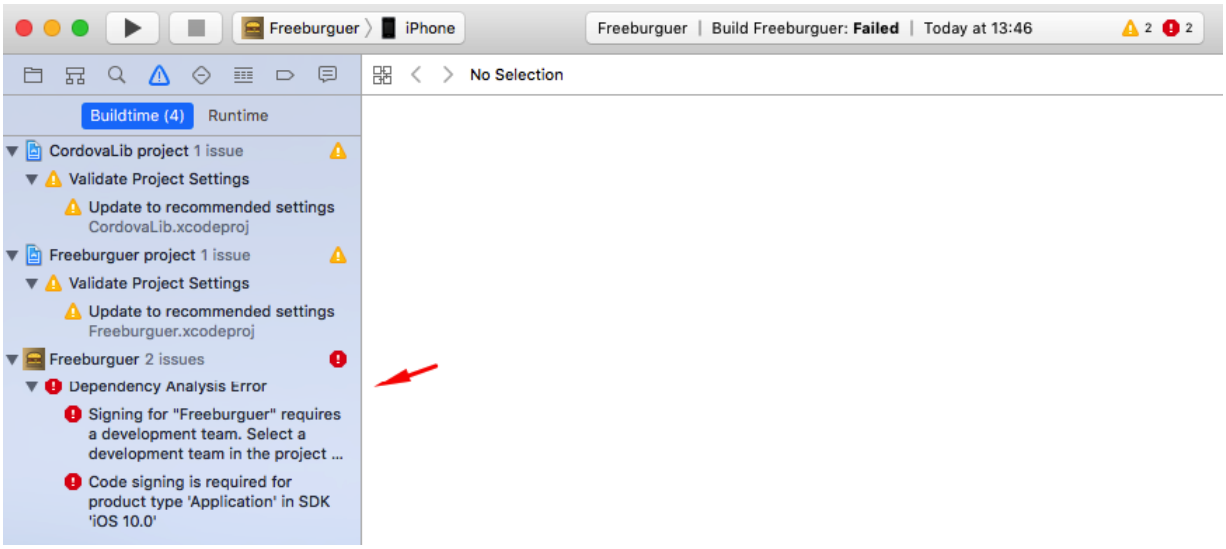

Freeburguer
iPhone
Freeburguer | Build Freeburguer: Failed | Today at 13:46
No Selection
Buildtime (4)
Runtime
CordovaLib project 1 issue
Validate Project Settings
Update to recommended settings
CordovaLib.xcodeproj
Freeburguer project 1 issue
Validate Project Settings
Update to recommended settings
Freeburguer.xcodeproj
Freeburguer 2 issues
Dependency Analysis Error
Signing for "Freeburguer" requires a development team. Select a development team in the project ...
Code signing is required for product type 'Application' in SDK 'iOS 10.0'

Figura 2.14: Erro de assinatura

Ao selecionar o nosso Apple ID (Personal Team), você vai perceber que o Xcode criará um *Signing Certificate* automaticamente.

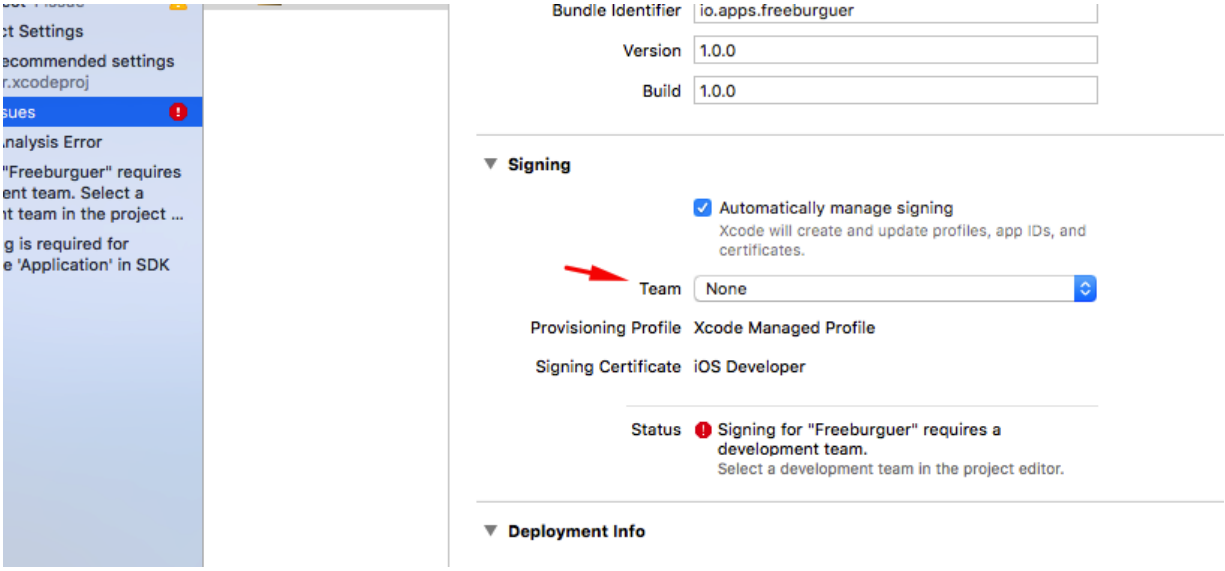


Figura 2.15: Seleção do Apple ID Team

Execute novamente o projeto, e teremos agora um alerta dizendo que precisamos autorizar no dispositivo aquele Apple ID como confiável. Então, poderemos realizar os testes e executar aplicativos diretamente do Xcode para o aparelho.

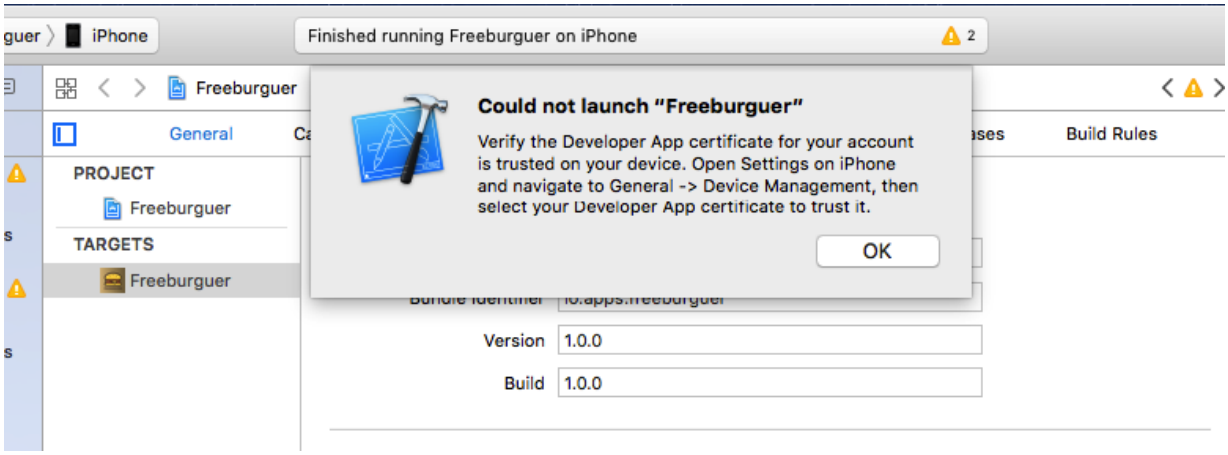


Figura 2.16: Alerta para confirmação de confiança do Apple ID no iPhone

Você pode reparar que o aplicativo já se encontra no aparelho, porém, ao tocá-lo, você receberá a seguinte mensagem:

Figura 2.17: Alerta de confiança do sistema

Para autorizar a nossa conta da Apple no iPhone, vá em `Ajustes -> Geral -> Gerenciamento de Dispositivo`. Dentro dessa janela, você vai

ver uma listagem de contas; é só pressionar sobre a sua conta – no nosso caso, *apple@diogomachado.com*.

Figura 2.18: Lista de Apple IDs

Dentro dessa nova janela, você verá um botão chamado `Confiar em apple@diogomachado.com` . Basta tocar e autorizar o Apple ID e, de agora em diante, o Xcode vai executar a aplicação diretamente no seu aparelho.

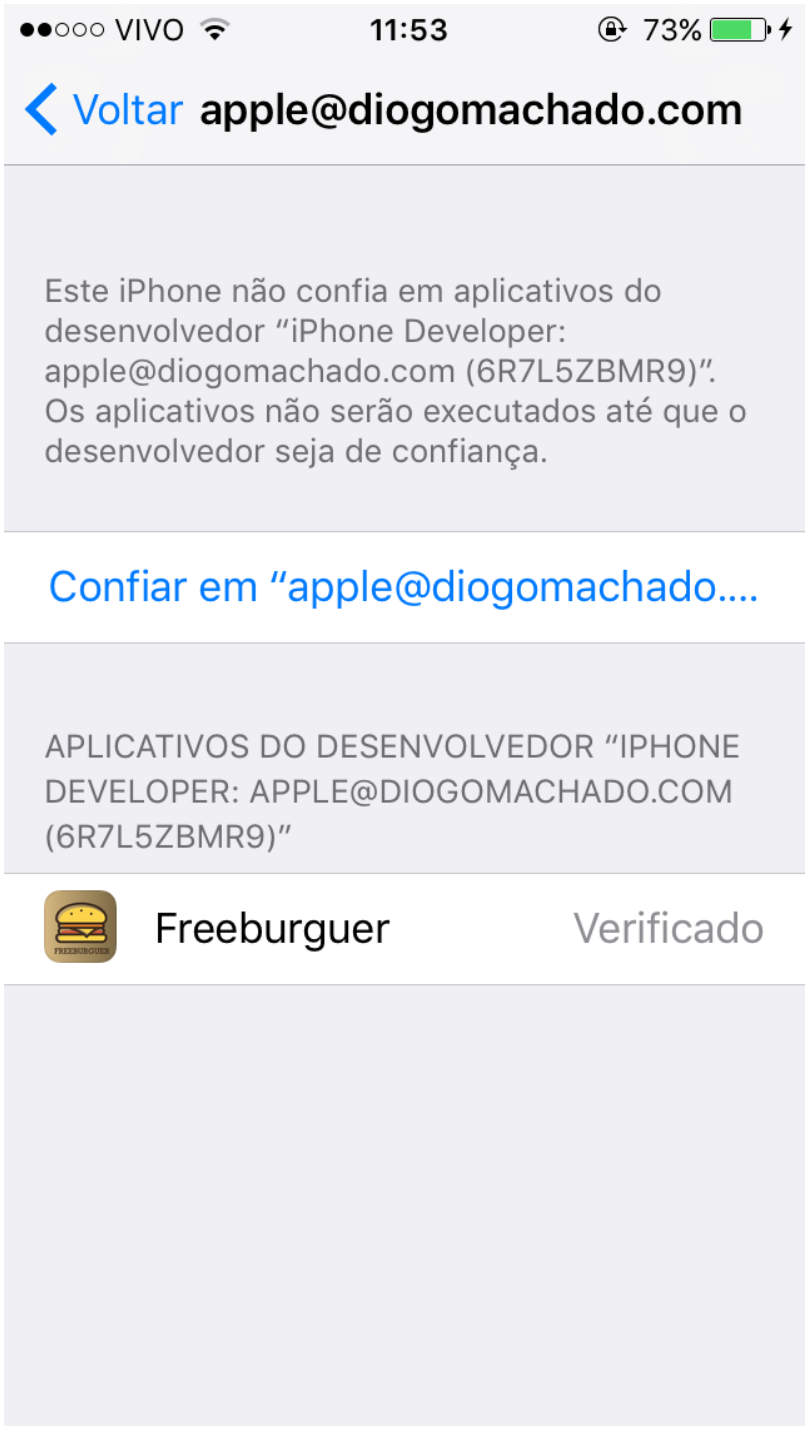

●●○○○ VIVO
11:53
73%
Voltar
apple@diogomachado.com
Este iPhone não confia em aplicativos do desenvolvedor “iPhone Developer: apple@diogomachado.com (6R7L5ZBMR9)”. Os aplicativos não serão executados até que o desenvolvedor seja de confiança.
Confiar em “apple@diogomachado....
APLICATIVOS DO DESENVOLVEDOR “IPHONE DEVELOPER: APPLE@DIOGOMACHADO.COM (6R7L5ZBMR9)”
Freeburguer
Verificado

Figura 2.19: Tela de confirmação de confiança

●●●○○ VIVO
11:53
73%
Voltar
apple@diogomachado.com
Este iPhone não confia em aplicativos do desenvolvedor "iPhone Developer: apple@diogomachado.com (6R7L5ZBMR9)".
Confiar em Aplicativos de "iPhone Developer: apple@diogomachado.com (6R7L5ZBMR9)" Neste iPhone
A confiança permitirá que qualquer aplicativo deste desenvolvedor seja usado no seu iPhone e pode permitir acesso aos seus dados.
Cancelar
Confiar

Figura 2.20: Alerta do sistema para confiar na conta

Você pode baixar o aplicativo pronto até aqui em http://bit.ly/cordova-livro. O projeto é o **freeburguer-cap2**.

2.8 Revisão

Neste capítulo, assumimos a responsabilidade de desenvolver o aplicativo Freeburguer. Antes de começar a criar as funcionalidades do aplicativo, fizemos todas as configurações iniciais no arquivo `config.xml`. Também ajustamos o plugin da StatusBar com uma aparência mais nativa, e geramos e configuramos os ícones e splashscreens das plataformas Android e iOS.

Partimos para o próximo capítulo com o objetivo de desenvolver de fato toda a UI (interface gráfica) e a UX (experiência do usuário) da aplicação. Mostraremos duas formas de se trabalhar design, uma usando um framework mobile e outra mostrando o caminho para desenvolver seu próprio layout mobile.

Capítulo 3
O design do aplicativo e a experiência do usuário

Poderíamos pular esta etapa se pensássemos em usar um framework como o Framework7, que oferta uma grande quantidade de componentes visuais prontos para uso. Porém, convido-o a fazer ao contrário nesse momento.

Vamos deixar a preguiça de lado e adentrar o design, de modo que você possa aprenda a *pensar* e *criar* do zero. Isso lhe habilitará a desenvolver qualquer aplicativo. Não estou dizendo que você nunca deve usar frameworks, mas, como desenvolvedor, você precisa avaliar a necessidade de cada projeto.

Isso porque importar arquivos JavaScript e CSS desses frameworks, apenas para usar um ou outro componente, terá um custo desnecessário para o browser. É mais valioso você saber criar um componente do que usar um pronto.

> Toda vez que falarmos de *componente* neste capítulo, entenda que estamos falando de um elemento visual que é resultado da combinação da marcação HTML, embelezado com CSS.

3.1 As tendências do design mobile

Este não é um livro de design, porém, é importante entendermos como o design de apps evoluiu e os seus conceitos básicos. Só assim é possível desenvolver e acompanhar o mercado de aplicativos.

Anos atrás, com o surgimento do primeiro iPhone, o design de aplicativos era voltado para um conceito chamado

Skeuomorphism. Na prática, o conceito buscava aproximar o design de ícones e elementos dentro do aplicativo de materiais, ou objetos do mundo real (como couro, pedra ou papel).

Figura 3.1: A logo do Instagram no passado usava o conceito de Skeuomorphism

Com o passar do tempo, o mercado (o Google com o Android, e a Microsoft com o Windows) começou a caminhar para a construção de uma interface mais simplista, limpa e de objetos planos, conceito conhecido como **Flat Design**. A Apple resistiu por mais tempo, porém também acompanhou o sucesso que o Flat Design vem tomando, e passou a adotar o conceito tanto para o iOS como para o MacOS.

Figura 3.2: Logo atual do Instagram utilizando o conceito Flat Design

Os designers do Google, ainda não satisfeitos, anunciaram há algum tempo uma documentação nova (hospedada em https://material.io) do que seria uma evolução do Flat Design, nomeada por eles de *Material Design*. Ela segue o princípio básico do Flat, mas incorpora outros elementos, como sombras.

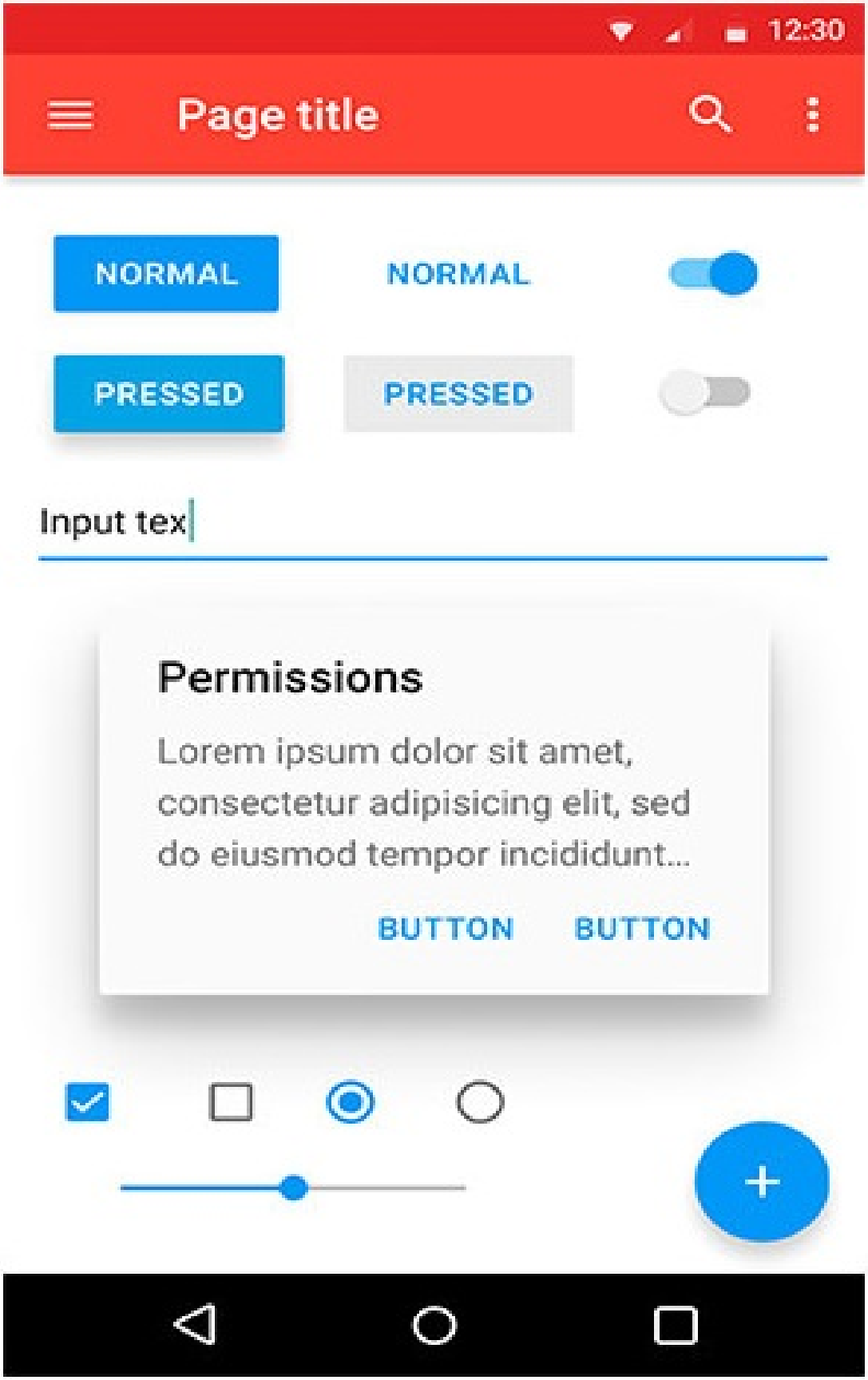

12:30
Page title
NORMAL
NORMAL
PRESSED
PRESSED
Input tex
Permissions
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt...
BUTTON
BUTTON

Figura 3.3: Exemplo de elementos do Material Design

É importante você conhecer essa evolução que o design tomou, pois ajuda na hora de escolher um ícone, um componente, ou até mesmo um framework. Digo isso porque existem frameworks que pararam no tempo em relação à interface mobile.

Os dois mais completos atualmente são o Ionic Framework e o Framework7, que oferecem uma interface multiplataforma bem atual, tanto para o modelo Flat (que o iOS usa) quanto para o Material Design do Android. Em resumo, o iOS segue a linha do Flat, e o Android do Material Design, que seria uma evolução do Flat, mas com suas particularidades adicionadas pelo Google.

3.2 A decisão: experiência nativa ou customizada

Não temos um kit de interface nativo na plataforma Cordova, e isso nos deixa livres para usar o CSS e a nossa criatividade como bem entendermos. Porém, o que deve ser analisado na hora de escolher um padrão a se seguir em um novo projeto é: se você deseja seguir uma experiência voltada para o nativo – ou seja, seguindo a criação da interface (ícones, fonte, componentes) usando as *guidelines* (diretrizes de uso) da plataforma; ou se toda a interface será customizada – opção preferida para jogos.

Experiência nativa

Não confunda a *interface nativa* com *experiência nativa*. Já sabemos que o Cordova não tem interface, portanto, não estamos falando da interface nativa e, sim, de criar uma experiência nativa usando CSS e HTML, seguindo as diretrizes de design das plataformas.

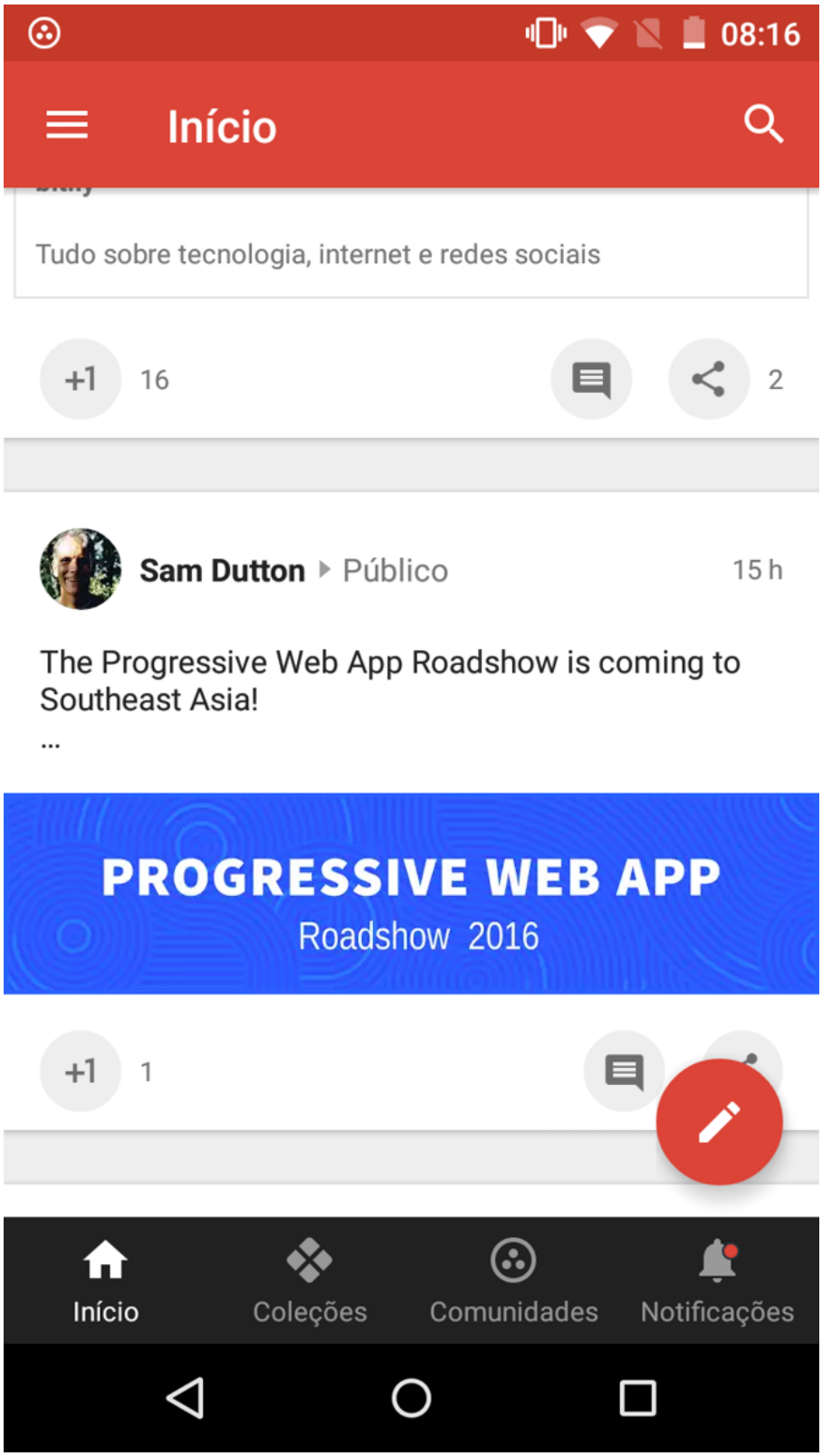


Figura 3.4: Interface nativa do aplicativo Google+

Pontos positivos:

- Favorece a experiência do usuário, que já está adaptado à interface do aparelho;
- Descomplica o desenvolvimento da interface, porque tem todo um guia já pronto para seguir, não precisa inventar nada a nível de design;
- Bom porque a maioria dos tipos de aplicativos encaixa-se bem no design nativo.

Pontos negativos:

- Ruim porque é mais trabalhoso diferenciar o layout para cada plataforma.

Existem características comuns em um design nativo de uma plataforma para a outra, e as mais notáveis são:

- Paleta de cores;
- Componentes (botões, alertas, galerias, barra de status);
- Tipografia;
- *Patterns* (padrões de exibição de erros, notificações, formato de datas e formas de escritas);
- Interações (animações, formatos de interação e usabilidade).

Todos esses itens abordados caracterizam de forma única o design nativo de uma plataforma. Cada sistema operacional tem suas diretrizes, e você pode aprofundar-se lendo os links a seguir:

- Diretrizes oficiais do Google – https://material.io/guidelines
- Diretrizes oficiais da Apple para o iOS – https://developer.apple.com/ios/human-interface-guidelines

Experiência customizada

Uma experiência customizada é quando produzimos uma interface única para aquele aplicativo. Um exemplo rápido são os jogos mobile. A maioria dos desenvolvedores escolhe produzir a interface

do aplicativo seguindo a temática do jogo, como no jogo *Build a Bridge*, mostrado na figura a seguir.

Figura 3.5: Interface customizada do jogo Build a Bridge para iOS

Pontos positivos:

- Favorece a criação de uma experiência única;
- Bom porque pode ser reutilizado em várias plataformas.

Pontos negativos:

- Ruim porque pode confundir o usuário que não está acostumado a usar aquele conjunto de interface;
- Nem todos os tipos de aplicativos encaixam-se bem, e a grande maioria dos jogos é o tipo de app que mais usa essa interface.

Em geral, você vai optar por essa opção na criação de jogos, ou quando o cliente já vier com um design pronto e desejar apenas que você desenvolva. Essa opção também é a mais rápida quando você deseja entregar um aplicativo híbrido para duas ou mais plataformas

(Android, iOS, Windows Phone) sem diferenciar o layout e os componentes.

Decisão híbrida de experiência nativa com customizado

É comum também ver uma fusão das duas ideias, ou seja, ver dentro de um aplicativo alguns elementos que seguiram o design nativo junto a elementos customizados que foram criados unicamente para ele.

Figura 3.6: Instagram do Android e iOS possui elementos do design nativo e também customizados, como a área de histórias, igual nas duas plataformas

Pontos positivos:

- Não foge totalmente do design que o usuário está acostumado;

- Agiliza o desenvolvimento, por não ter de diferenciar detalhes em cada plataforma.

Pontos negativos:

- Geralmente, o design terá mais traços de uma plataforma. Se você deseja entregar uma experiência diferente em cada uma, essa decisão não é a mais adequada.

Nossa escolha

Você poderia optar por qualquer um dos três direcionamentos de design que citei, cada um com suas vantagens e desvantagens a *nível de design*. Essa escolha não afetará as funcionalidades do seu aplicativo, apenas ditará qual design será entregue e como você e sua equipe vão planejar o desenvolvimento dos elementos do app.

Saber qual direção será tomada é importante logo no início do projeto, até mesmo para você poder ter noção se poderá utilizar alguma solução pronta, como Ionic, ou se terá de codar todo o layout. No caso do Freeburguer, decidimos seguir a linha do **design nativo**, porque o cliente pediu que a plataforma seja o mais próximo possível de um aplicativo nativo, feito com a linguagem Android ou Swift.

Essa escolha é boa, pois, além dos benefícios que falamos, também demonstra um pouco da qualidade do aplicativo. O que quero dizer é que, com a popularização do PhoneGap e do Cordova, muitos apps ruins de design foram desenvolvidos e publicados. Entretanto, o design é um grande divisor entre bom e ruim ao olhar básico de um usuário comum, que acabou de abrir o seu app. Quantas vezes você já se surpreendeu ao abrir um novo aplicativo?

3.3 O tamanho de um pixel em diferentes resoluções de tela

Se fôssemos customizar um botão agora, poderíamos fazer desta forma:

```
button{
    height: 44px;
}
```

Isso está errado? Não! O que acontece é que existe uma diferença entre o **CSS Pixel** e o **pixel físico** que um aparelho possui.

Logo, concluímos rapidamente que esse mesmo botão pode apresentar um tamanho relativamente menor em uma tela com **densidade** maior, ou ao contrário. Essa densidade é fortemente influenciada pela quantidade de pixels que cabem em uma polegada de tela, também conhecido por **ppi** (pixel por polegada ou **ppp**, em português).

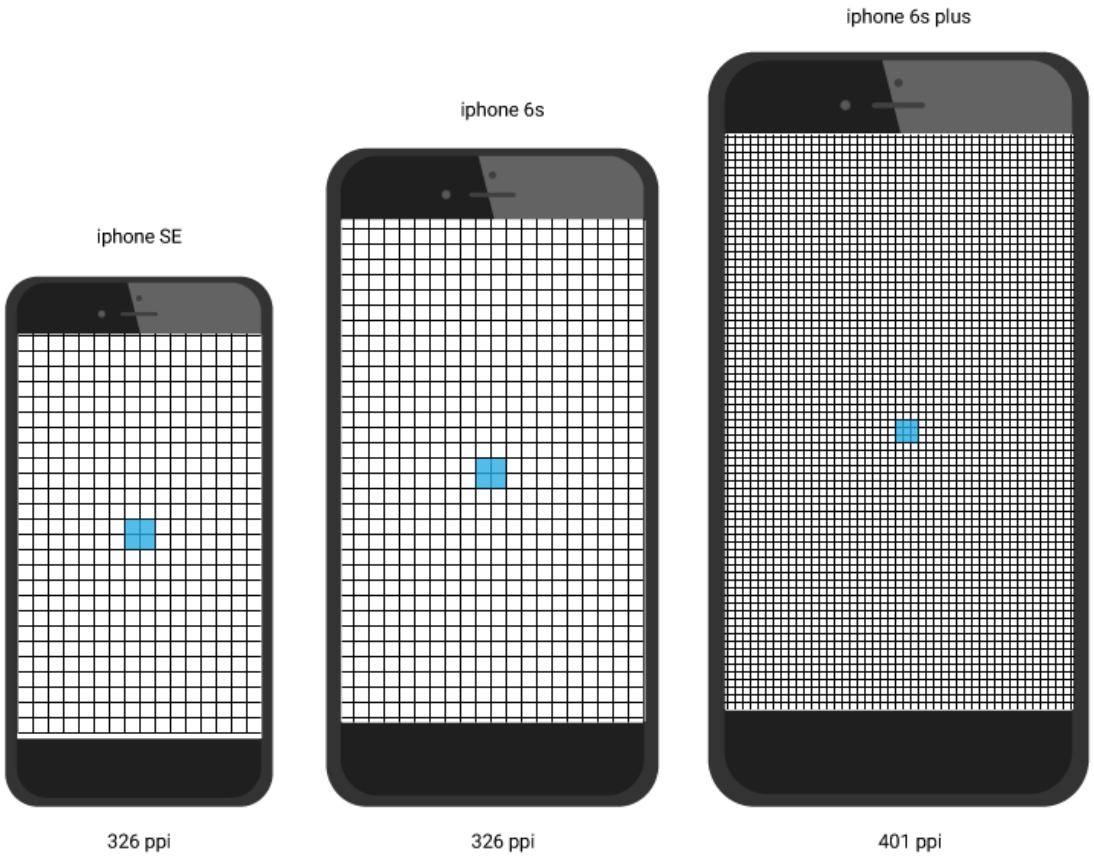


Figura 3.7: A área mais escura representa figurativamente uma polegada

É importante saber que, para os dispositivos, um pixel em si não tem valor ou representação física. Ele só pode transportar valor por meio da sua relação com o tamanho físico da tela, criando a resolução – ou PPI, como explicamos anteriormente.

Media queries e a densidade da tela

A chave para fazer a diferenciação de alguns elementos do design é o *media queries*. Com ele, faremos os ajustes necessários para cada dispositivo que quisermos otimizar o layout.

Device pixel ratio	Densidade generalizada da tela	Pontos por polegada CSS (dpi)	Pontos por CSS pixel (dppx)

Device pixel ratio	Densidade generalizada da tela	Pontos por polegada CSS (dpi)	Pontos por CSS pixel (dppx)
1x	MDPI	96dpi	1dppx
1.5x	HDPI	144dpi	1.5dppx
2	XHDPI	192dpi	2dppx

A tabela anterior mostra a relação entre dpi e dppx. Podemos dizer que 1dppx é equivalente a 96dpi, 2dppx é igual a 192dpi e assim por diante. Você pode usar tanto a unidade dpi quanto a dppx para especificar o seu CSS:

```
@media screen and (min-resolution: 300dpi) { ... }
```

Também sobre a tabela anterior, podemos ver a relação de igualdade que o Device Pixel Ratio (DPR) tem com a unidade dppx, portanto, **1x é igual a 1dppx**. É bastante útil saber disso, já que podemos testar o design do nosso aplicativo no modo responsivo do Google Chrome:

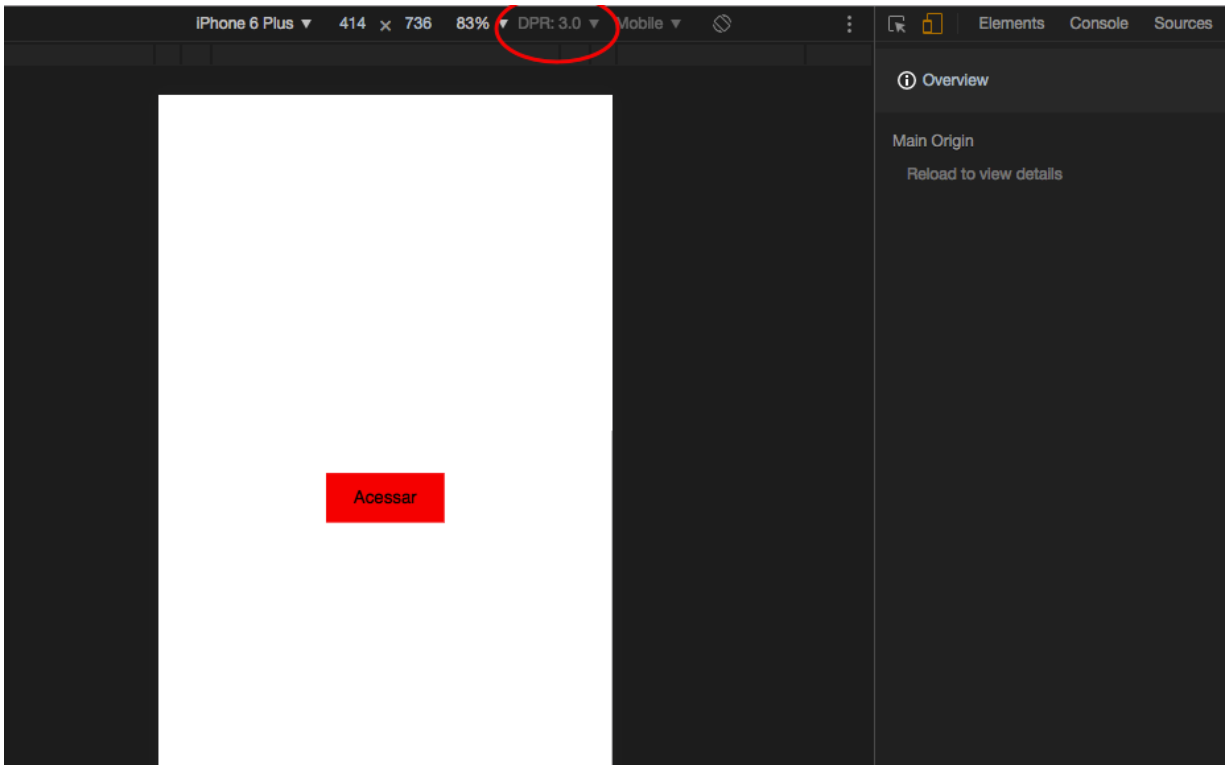


Figura 3.8: Modo responsivo ativado no navegador Google Chrome

Portanto, para customizarmos um componente usando Media Queries no iPhone 6 Plus (que tem um DPR 3), poderíamos fazer como o código a seguir:

```
@media screen and (min-resolution: 3dppx) { ... }
```

Qual a diferença de DPI para PPI?

É comum confundir essas duas nomenclaturas. O que acontece é que *DPI (Pontos por polegada)* é uma medida de ponto espacial inicialmente usada em impressões. É o número de gotas de tinta que uma impressora coloca em uma polegada – ou seja, quanto mais pontos por polegada, mais nítida será a sua imagem. O mesmo conceito é aplicado para telas de computadores com o nome de *PPI (Pixels por polegada)*, seguindo o mesmo princípio. A medida é referente a quantos pixels a tela exibe por polegada.

Se você está se perguntando como saber tantas especificações, acalme-se. Existem alguns sites que podem nos ajudar nessa tarefa:

- Guia do Google de tamanhos e resoluções de múltiplos dispositivos – https://material.io/devices
- Outra opção de guia de tamanhos – http://screensiz.es/phone
- Outra opção bacana sobre resoluções – http://dpi.lv/

E é claro, a dica é sempre começar com calma e ir otimizando o design do aplicativo a cada atualização, pois é bem difícil começar um app perfeito. Vale testar e saber fazer essas otimizações. Marque alguns breakpoints comuns e, depois, vá otimizando de acordo com as suas necessidades.

> **Quer saber mais sobre DPI e densidade de telas?**
>
> Um designer chamado *Peter Nowell* escreveu um artigo bem completo em sua página no site Medium, tem até um vídeo explicativo. Apesar de estar em inglês, vale muito a pena a sua leitura: http://bit.ly/2IY6UcR.

3.4 Preparando o aplicativo para construir o design

Poderíamos muito bem usar um framework de interface como o *Framework7*, *Goratchet*, *Ionic* ou *Angular Material*. Porém, essas ferramentas trazem coisas prontas e aqui escolhemos construir do zero justamente para aprender a fazer. Frameworks são bons, mas nem sempre precisamos deles para inchar nossa aplicação.

Para o design, precisamos especificar e construir os seguintes componentes:

- Toolbars
- Tipografia
- Botões
- Ícones
- Paleta de cores
- Alertas
- Sistema de grid (layout)

Um dos motivos de escolhermos usar um framework SPA é por podermos utilizar um sistema de rotas robusto. Como estamos usando AngularJS, vamos usar a dependência `ngRoute` para definir as rotas necessárias para o aplicativo Cordova.

Como o `ngRoute` é um componente do AngularJS, ele é importado no script `angular-route.min.js`, definido no arquivo principal da aplicação `index.html`. Esse script já estava incluso no template que usamos para iniciar o projeto no capítulo anterior, mas poderia ser encontrado no link https://code.angularjs.org/1.6.6.

As rotas estarão definidas dentro do arquivo `bootstrap.js`, mais especificamente na função `.config`, utilizando o objeto `$routeProvider`. A cada `when`, definimos uma rota diferente e, em último caso, caímos em `otherwise`, que é a rota padrão.

```
angular.module('app',['ngRoute'])
.config(function($routeProvider)
{
        $routeProvider
        .when('/', {
            templateUrl  : 'app/views/home.html'
        })
        .when('/cardapio', {
            templateUrl  : 'app/views/cardapio.html'
        })
        .when('/buscar-pedido', {
            templateUrl  : 'app/views/pedido-busca.html'
        })
        .when('/pedido-info', {
            templateUrl  : 'app/views/pedido-info.html'
```

```
        })
        .otherwise ({ redirectTo: '/' });
});
```

É interessante você entender como funciona o esquema das *Views* no AngularJS usando rotas. A primeira coisa que fazemos é marcar no arquivo `index.html` uma `div` com o parâmetro `ng-view` :

```
<div ng-view></html>
```

Feito a marcação do `ng-view` , a cada troca de rota, o framework SPA vai injetar a *View* definida em `templateUrl` , no arquivo `bootstrap.js` especificado para aquela rota.

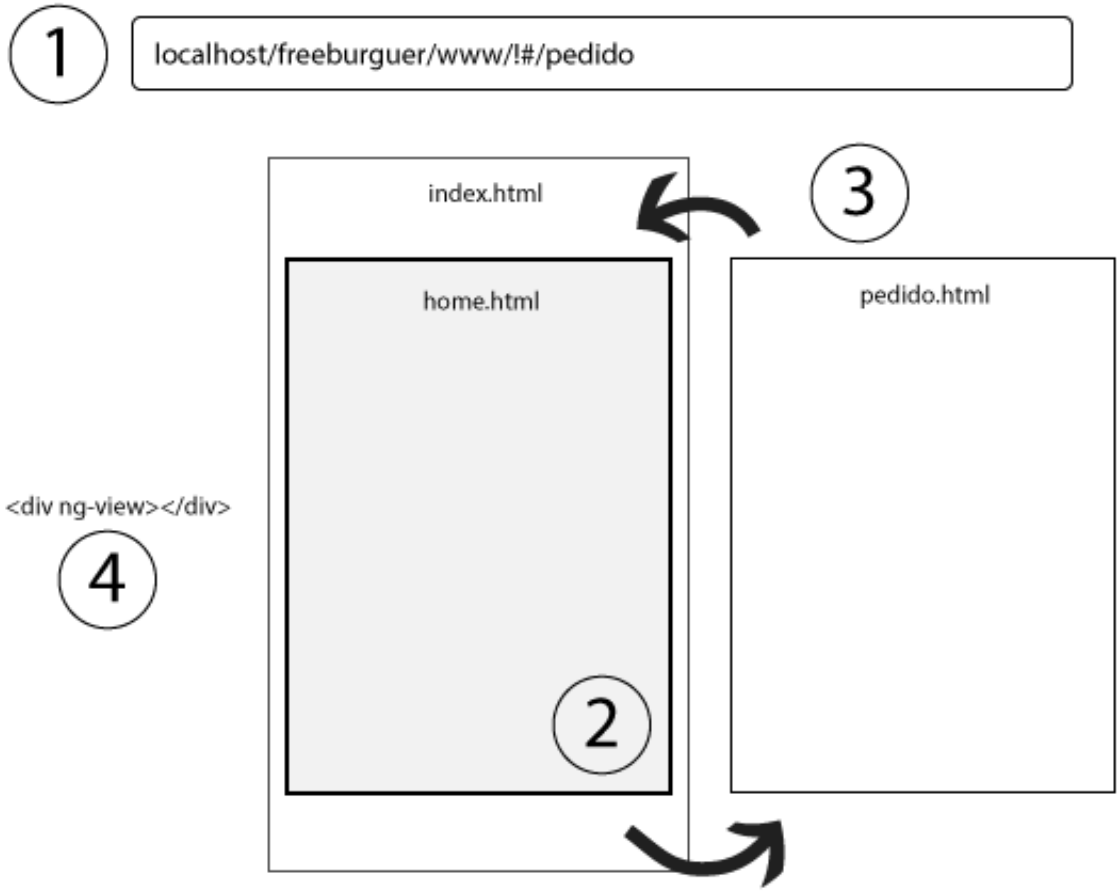


Figura 3.9: Funcionamento das Views no AngularJS

Na imagem anterior, imagine que você trocou para a rota `/pedido` no navegador (1). Logo, o AngularJS vai tirar o *pedaço* da View que está injetada (2) e, em seguida, injetar o *pedaço* da View correta para aquela rota (3) na `div` especificada, com `ng-view` (4) – que está em `index.html` em nosso caso.

Daqui em diante, desenvolveremos o layout do aplicativo baseado nas telas que vamos conhecer a seguir. É importante explicar que não mostrarei todo o código das *Views*, pois, em alguns casos, trata-se apenas da construção de uma página HTML com CSS. Claro que vou destacar os pontos mais importantes, e você também pode baixar o projeto pronto deste capítulo em http://bit.ly/cordova-livro, para acompanhar e ver como foi feito.

Os arquivos das *Views* foram criadas com o nome igual ao que definimos nas rotas. Também existe dois fragmentos de HTML, que chamei de `_svg-home.html` e `_svg-pedido-busca.html`. Esses dois arquivos são incluídos na `home.html` e na `pedido-busca.html`, de forma dinâmica, com um recurso do AngularJS chamado `ng-include`. Fiz isso para deixar a *View* com a lógica do HTML mais limpa, pois o código do SVG é bem extenso.

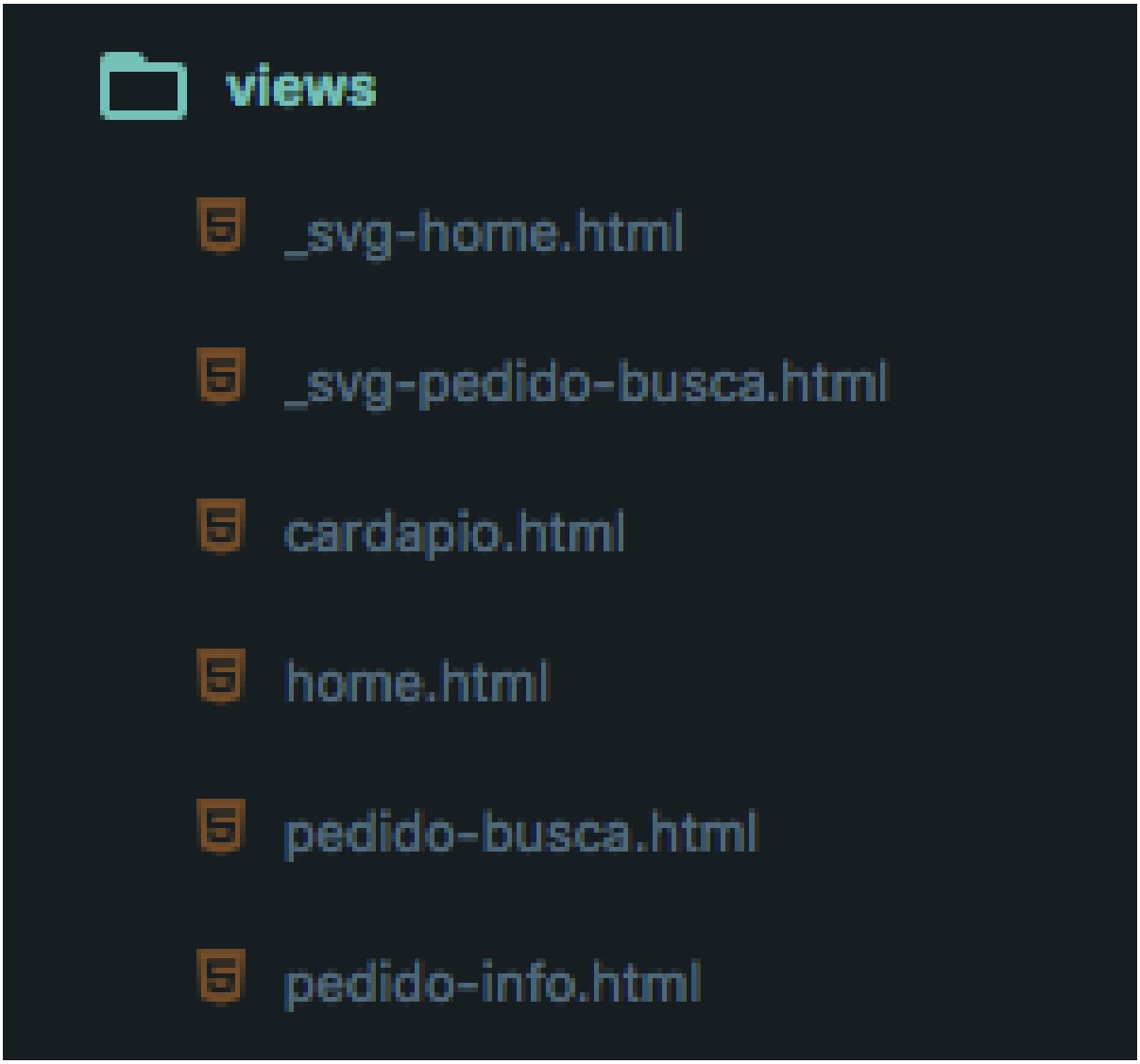


Figura 3.10: Organização das Views

O código com o `ng-include` na `home.html` se parece com o mostrado a seguir:

```
<div class="content home content-center">
    <div class="svg" ng-include="'app/views/_svg-home.html'"></div>

    <!-- Restante do conteúdo ...-->
</div>
```

E o da página `pedido-busca.html` :

```
<div class="content home content-center">
    <div class="svg" ng-include="'app/views/_svg-pedido-busca.html'">
</div>

        <!-- Restante do conteúdo ...-->
</div>
```

Diferenças no layout com CSS

Como escolhemos trabalhar com um design voltado ao nativo, uma estratégia que uso é diferenciar o layout usando classes CSS com o nome do sistema, como `.android` e `.ios`. Assim, podemos construir um componente e diferenciar os detalhes com essas classes.

Para saber em qual sistema a nossa aplicação está trabalhando ao ser carregada, vamos usar o plugin `cordova-plugin-device`. Ele tem uma função que retorna o nome da plataforma em que o Cordova está, muito útil neste momento.

Dentro da raiz do diretório do aplicativo, adicione o plugin ao projeto:

```
cordova plugin add cordova-plugin-device --save
```

Agora na aplicação, vamos abrir o arquivo `bootstrap.js`. Na nossa organização, este é responsável por fazer justamente essas inicializações. No arquivo, olharemos para dentro de `.run`; no AngularJS, isso quer dizer que o código ali dentro será executado logo após o carregamento do framework, por isso essa área é muito útil para fazer inicializações na aplicação.

Também vamos precisar do `$rootScope`, que vai expor a nossa variável Angular globalmente na aplicação. Ou seja, teremos acesso à variável `$rootScope.device` em qualquer lugar do código, até mesmo em outros controladores.

Algo comum de fazermos quando estamos trabalhando com Cordova é identificar quando o framework está carregado. Para isso, temos no nosso código o seguinte *listener*:

```
document.addEventListener("deviceready", function () {
    // Cordova inicializado
}, false);
```

Sabendo que o Cordova e o AngularJS estão 100% carregados, podemos capturar o `device` que estamos de forma simples:

```
document.addEventListener("deviceready", function () {
    var plataforma = device.platform;
    $rootScope.device = plataforma.toLowerCase();
}, false);
```

Injeção de dependências do AngularJS

Sempre que for usar um recurso do AngularJS, você precisa injetá-lo. No caso do método `.run()`, faremos como o código a seguir:

```
.run(function($rootScope)){
}
```

Com essa variável `device`, podemos customizar os `.html` das *Views* da nossa aplicação juntamente com as folhas de estilo CSS e entregar um layout diferenciado no Android e no iOS – ou em qualquer outra plataforma que o Cordova suporte.

Faça um teste usando o seu aparelho físico, conforme explicado no capítulo anterior, e emita um `alert()` para ver tudo funcionando:

```
document.addEventListener("deviceready", function () {
    console.info("Cordova inicializado com sucesso.");

    var plataforma = device.platform;
    $rootScope.device = plataforma.toLowerCase();

    alert($rootScope.device);
}, false);
```

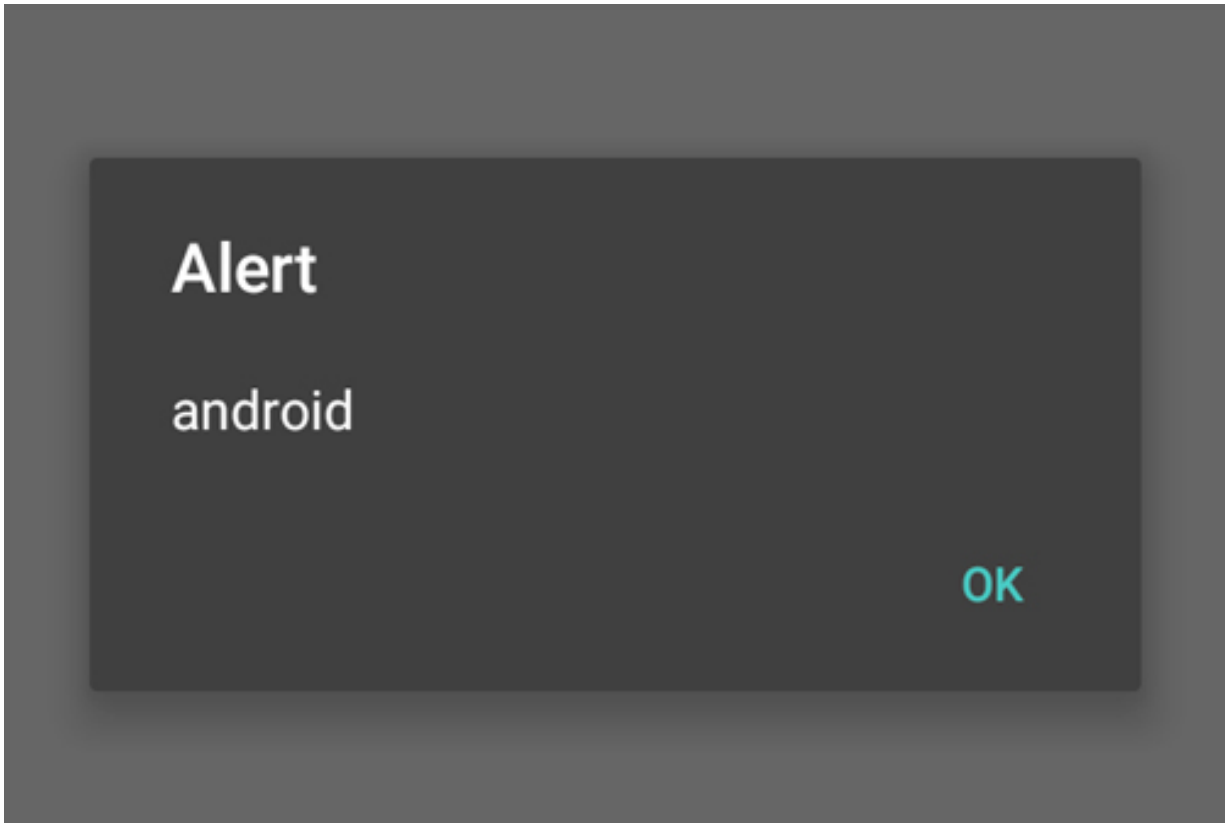


Figura 3.11: Resultado do comando Alert

3.5 As telas do aplicativo

Vamos agora conhecer as telas das *Views*, definidas nas rotas, que vamos construir.

> O protótipo do aplicativo Freeburguer foi feito no *Adobe CC XD*, e você pode baixá-lo em: http://bit.ly/2b6nA8m.

Home

A *Home* é a tela principal do aplicativo, em que o nosso usuário vai:

1. Abrir e ler o QRCode ao tocar no ícone, carregando o cardápio;

2. Encontrar a empresa inserindo direto o seu código;
3. Ir para um pedido específico já realizado ao clicar no botão `Meus pedidos`.

12:30
código da casa
Meus pedidos

Figura 3.12: Imagem ilustrativa da Home

Quando o usuário tocar no `input` para inserir o código da empresa, o teclado será exibido empurrando o botão `Meus pedidos` para cima, já que isso vai atrapalhar o usuário que estiver em telas pequenas:

Figura 3.13: Botão Meus pedidos visível com teclado aberto

Vamos corrigir isso com o CSS a seguir, adicionado no arquivo `freeburguer -> css -> app.css` :

```
@media screen and (max-height: 340px){
    .footer{
        display: none;
    }
}
```

O arquivo da *View* será o `www -> app -> views -> home.html .`

Pedidos

A tela `Pedidos` é onde o usuário buscará por um pedido específico. Ele insere o código do pedido e é direcionado até as opções de pedido.

12:30

código do pedido

Voltar

Figura 3.14: Tela de buscar pedidos

O arquivo da *View* será o `www -> app -> views -> pedido-busca.html .`

Cardápio e seleção

O `Cardápio` é o local onde carregamos os itens que a empresa vende. Nele, o usuário vai poder:

1. Selecionar os itens e exibir o resumo do total;
2. Confirmar o fechamento do pedido e, em seguida, ser redirecionado à tela de acompanhamento e opções do pedido.

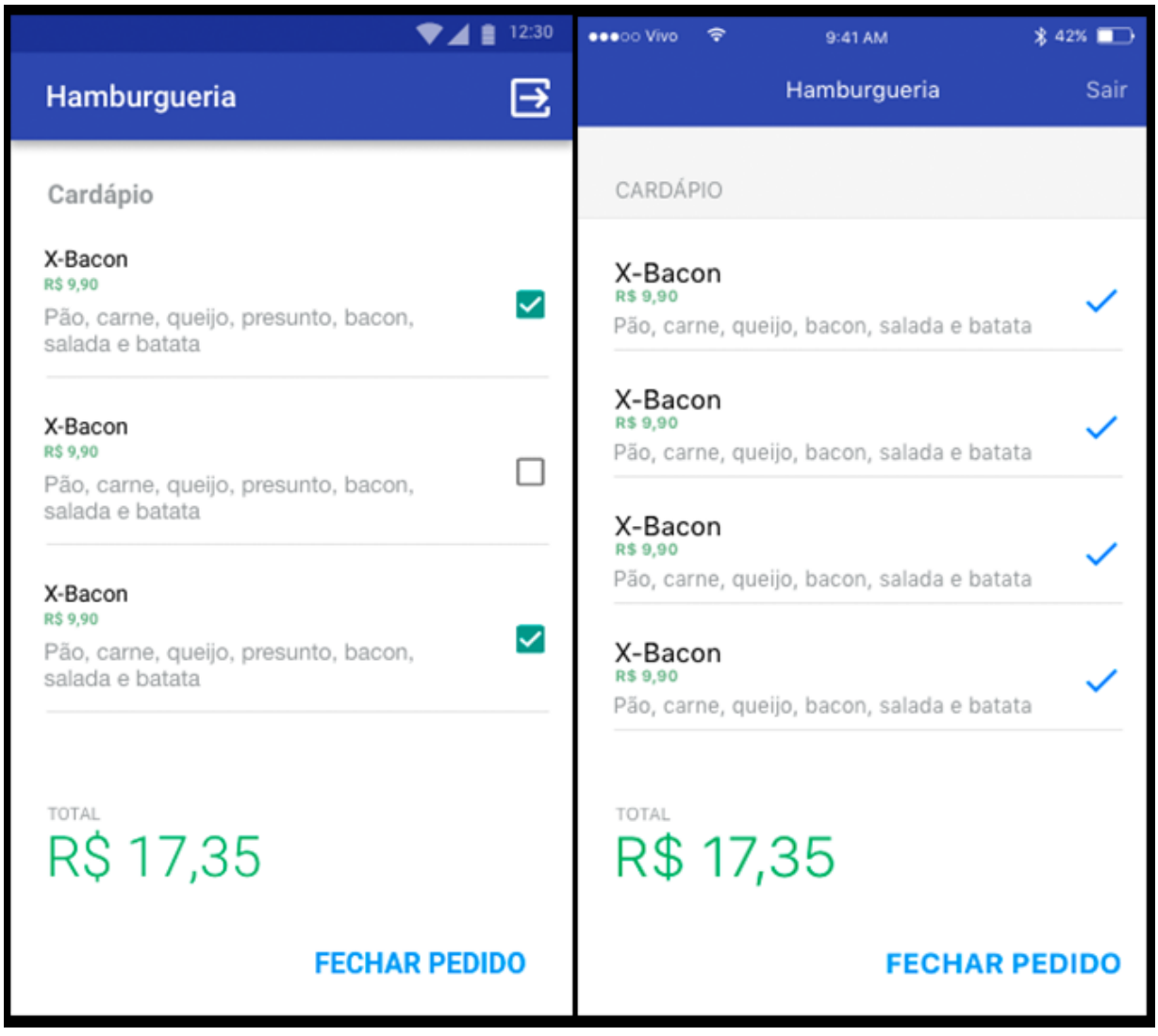


Figura 3.15: Tela de cardápio – Android e iOS

O arquivo da *View* será o `www -> app -> views -> cardapio.html .`

Informações do pedido

Essa é a tela final, na qual o app dá algumas opções ao cliente, que são:

1. Visualização dos dados básicos do pedido (código e valor total);
2. Opção de rastrear o pedido via coordenadas do GPS;
3. Opção de cadastrar o telefone da empresa na lista de contatos.

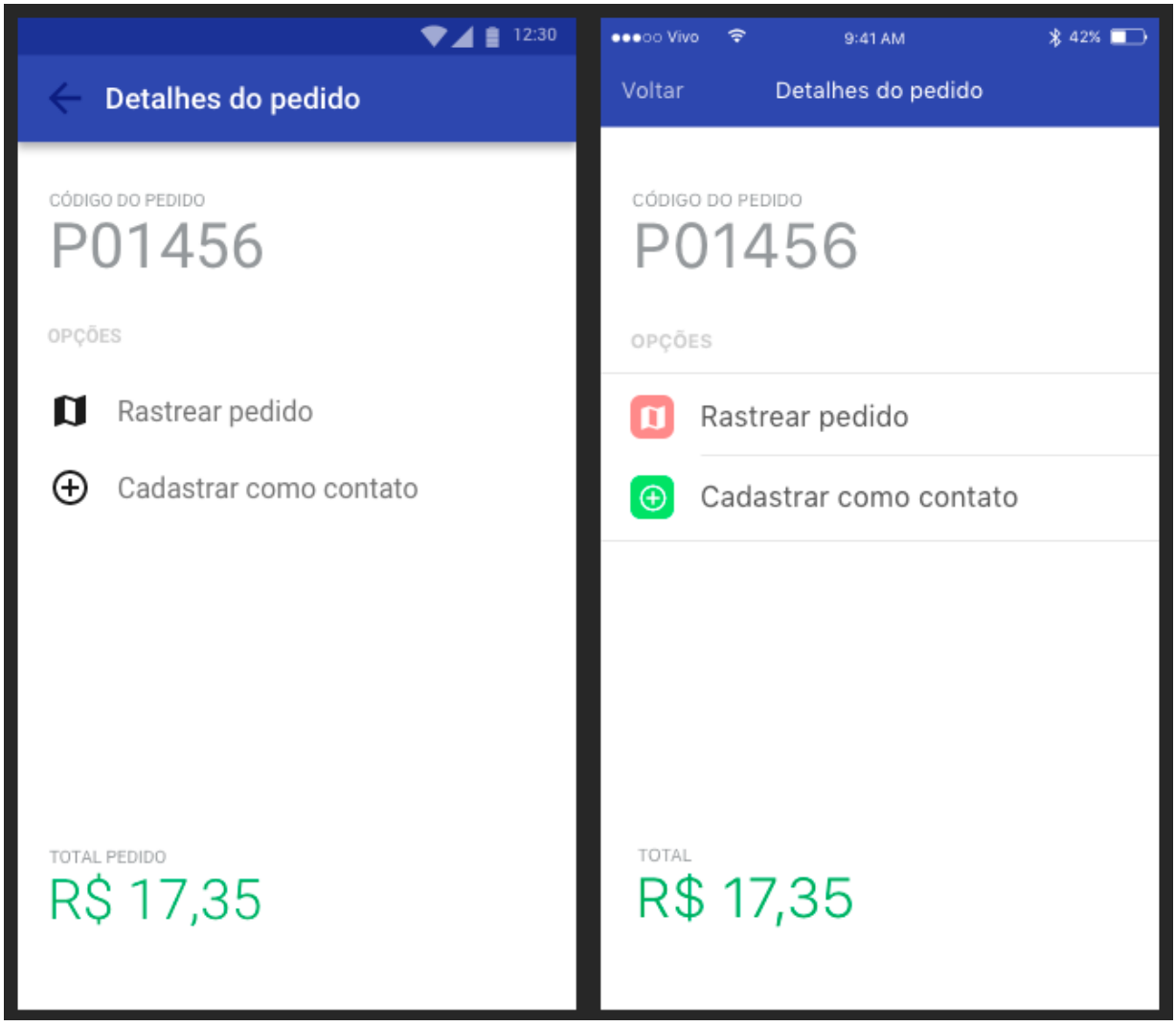


Figura 3.16: Tela do pedido – Android e iOS

O arquivo da *View* será o `www -> app -> views -> pedido-info.html .`

Por motivos didáticos, não vou explicar todo o código HTML das *Views*. Mas, para acompanhar, você já pode baixar o projeto completo deste capítulo no link http://bit.ly/cordova-livro.

3.6 A barra de ferramentas Toolbar

Uma *Toolbar* (barra de ferramentas) é um local usado no layout de uma aplicação para ações. Em aplicativos móveis, uma Toolbar é útil para informar onde o usuário se encontra, por exemplo, e também para ofertar botões de ação.

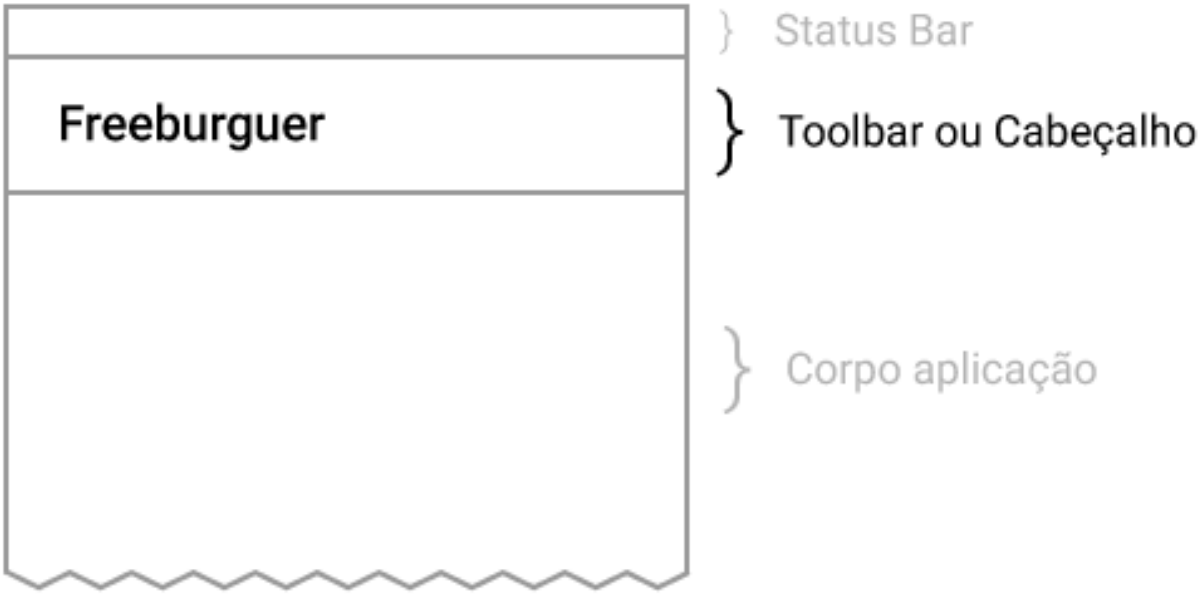


Figura 3.17: Anatomia da aplicação – Toolbar

O *Toolbar* que vamos construir será um componente marcado por HTML e estilizado com CSS, usaremos o recurso de *Media Queries* para dar comportamento diferente na classe dependendo da tela do aparelho, claro que esse é um trabalho que deve ser constante e nunca será perfeito.

Para o tratamento da altura do *Toolbar* vamos especificar alguns DPRs diferentes, você pode descobrir o DPR dos dispositivos que deseja adaptar o layout nos sites:

- Seção *Density* – https://material.io/devices
- Seção *PX Density* – http://screensiz.es/phone

O código a seguir é um pedaço do que faremos no CSS `www -> assets -> css -> app.css`. Por motivos didáticos, não colocarei todo o código do *Toolbar* aqui, mas você pode baixar o projeto para acompanhar o capítulo.

Para iPhone 4, iPhone 5 e iPhone 6, por exemplo, especificamos a altura do `.toolbar` com `50px`:

```
@media screen and (min-resolution: 2dppx) {
    .toolbar{
        height: 50px;
    }
}
```

Aparelhos com densidade ainda maior de tela, como o iPhone 6 Plus, especificamos a altura em `58px`:

```
@media screen and (min-resolution: 3dppx) {
    .toolbar{
        height: 58px;
    }
}
```

E para fechar, aparelhos com ainda mais densos em pixels (como o Nexus 6), especificamos `60px` de altura:

```
@media screen and (min-resolution: 3.5dppx) {
    .toolbar{
        height: 60px;
    }
}
```

Logo, cada aparelho terá um comportamento de altura diferente quando acessar a aplicação, melhorando a usabilidade e a

capacidade de adaptação do seu aplicativo.

O *Toolbar* terá dois tipos de comportamento diferentes, quando estiver em ambiente Android e quando estiver no iOS. Como a classe `.toolbar` já atende bem às cores para o iOS, então só diferenciaremos os detalhes no Android.

Repare no código a seguir que, sempre que aparecer uma classe `.android` adicionada dinamicamente pelo AngularJS (com a ajuda da variável `device`) e capturada no arquivo `bootstrap.js` em união à classe `.toolbar`, a *Toolbar* terá uma leve sombra, fazendo referência mais próxima ao design nativo do Android.

```
.toolbar.android{
    box-shadow: 0 1px 8px rgba(0,0,0,.3);
}
```

Agora vamos marcar o HTML das *Views* (visualizações) da aplicação com o código do *Toolbar*. Das *Views* que temos, duas terão *Toolbar*: a de seleção no cardápio `cardapio.html` e a de informações do pedido `pedido-info.html`.

Figura 3.18: Telas que usam Toolbar no projeto

Para o `cardapio.html`, definimos:

```
<header class="toolbar {{ device }} barFixed">
    <section class="buttons"></section>

    <section class="title {{ device }}">
                Hamburgueria
    </section>

    <section class="buttons">
        <button type="button" class="{{ device }}" ng-click="sair()">
            <i class="close"></i>
            <span class="text">Sair</span>
```

```
        </button>
    </section>
</header>
```

Para o `pedido-info.html`, definimos:

```
<header class="toolbar {{ device }} barFixed">
    <section class="buttons">
        <button type="button" class="{{ device }}" ng-click="ir('/buscar-
pedido')">
            <i class="icon-back"></i>
            <span class="text">Voltar</span>
        </button>
    </section>
    <section class="title {{ device }}">
        Detalhes do pedido
    </section>
    <section class="buttons"></section>
</header>
```

O que é a classe .barFixed?

Essa classe é, na verdade, um recurso do AngularJS chamado **diretiva**. Ela foi criada para calcular de forma dinâmica a altura do *Toolbar* deixando o elemento fixo. As diretivas são um recurso poderoso do AngularJS, e você pode conferir o código em `app -> diretivas -> barFixed.js`.

Repare que a tag `header` tem uma marcação especial do AngularJS `{{ device }}`, que é aquela variável que criamos na inicialização. Podemos e vamos usá-la em vários locais da marcação como forma de nos auxiliar na tarefa de diferenciação do layout.

Quais são as diferenças de design no iOS e no Android?

Uma dica é sempre ler os *guidelines* das plataformas. No iOS, geralmente o título da *Toolbar* é centralizado; já no Android, ele fica alocado à esquerda. O mesmo acontece com os ícones: no iOS, o mais comum é usar botões como texto, ou texto mais ícone; e no Android, a preferência é por ícones.

Nessa etapa de layout, vamos realizar muitos testes. Uma dica é usar bastante o aplicativo *PhoneGap Developer App*. Ele não é perfeito, mas funciona bem para essa etapa de prototipação do layout e, o melhor, funciona em tempo real. Basta salvar o arquivo CSS que está editando e você verá as alterações no dispositivo conectado. Acesse o site https://phonegap.com/products/#mobile-app-section, e baixe o aplicativo no seu celular de teste.

Para disponibilizar o seu aplicativo, basta acessar a raiz do projeto com o terminal e digitar `phonegap serve`. Após o comando um IP (*Internet Protocol*) será disponibilizado no terminal (como na imagem a seguir), e você precisará inseri-lo no aplicativo que acabou de baixar. Em seguida, seu app em desenvolvimento será baixado e executado, e toda alteração no código refletirá nele, facilitando a etapa da construção da interface.

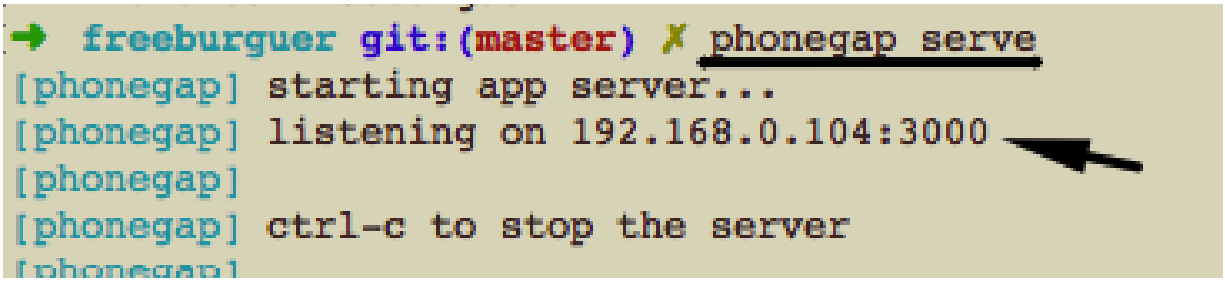

freeburguer git:(master) ✗ phonegap serve
[phonegap] starting app server...
[phonegap] listening on 192.168.0.104:3000
[phonegap]
[phonegap] ctrl-c to stop the server

Figura 3.19: Terminal executando o comando phonegap serve

Outra opção é testar o projeto diretamente no navegador. Se você roda um servidor *Apache* local, por exemplo, você pode acessar o diretório `www` da aplicação e vê-lo rodando. Porém, neste caso, não

cairemos no evento `deviceready`, porque não estamos em um ambiente nativo. Sempre que testar no navegador, lembre-se de retirar a lógica de dentro do `deviceready`:

```
$rootScope.device = "android";

document.addEventListener("deviceready", function () {
  // Código estaria aqui!
}, false);
```

3.7 Tipografia correta para cada plataforma

A tipografia é algo bastante relevante quando se tem como objetivo chegar o mais próximo da interface nativa. O que quero explicar é que cada plataforma adota uma família de fonte; podemos chamar isso de tipografia nativa. No Android, é usada a família *Roboto*; diferente do iOS, que recentemente adotou a fonte *San Francisco*.

Sabendo que temos essa diferença, novamente com a ajuda da variável `$rootScope.device`, vamos diferenciar as fontes. Começamos por defini-las no CSS via `@font-face`. Mas, antes de fazer isso, é importante saber que, às vezes, só baixamos a fonte em um formato, mas, para criar um `@font-face` mais abrangente para as plataformas do aplicativo, é legal definirmos um suporte para mais de um formato.

Os formatos aceitos são:

- WOFF 2.0 para os navegadores com suporte;
- WOFF para a maioria dos navegadores;
- TTF para navegadores Android antigos (anteriores à versão 4.4);
- EOT a navegadores IE antigos (anteriores ao IE9).

> Também existe o contêiner de fontes SVG, mas ele não é compatível com IE ou Firefox e foi desativado no Google Chrome. Assim, seu uso é limitado e não vamos considerá-lo.

Conhecendo esses formatos, os que nos interessam são WOFF 2.0, WOFF e TTF. Um site muito bom para gerar fontes em vários formatos é o *Font Squirrel* (https://www.fontsquirrel.com/tools/webfont-generator). Nele, no modo `Generator`, ao escolhermos a opção `Expert`, é possível gerar a fonte em vários formatos.

Ao fazer o download da fonte gerada, também ganhamos o CSS do `@font-face` de *brinde*, o que ajuda a poupar tempo. Já comentei que a fonte no Android é a da família *Roboto*, mas existem muitas variações dela, como podemos a seguir:

Figura 3.20: Família de fontes Roboto

Vamos usar duas variações da Roboto, a **Roboto-Regular** e a **Roboto-Bold**. Depois de baixar e extraí-las para `assets -> fonts`, inserimos o código do `@font-face`.

```
@font-face {
    font-family: 'Roboto-Regular';
```

```
    src: url('../fonts/roboto-regular-webfont.eot');
    src: url('../fonts/roboto-regular-webfont.eot?#iefix')
format('embedded-opentype'),
         url('../fonts/roboto-regular-webfont.woff2') format('woff2'),
         url('../fonts/roboto-regular-webfont.woff') format('woff'),
         url('../fonts/roboto-regular-webfont.ttf') format('truetype');
    font-weight: normal;
    font-style: normal;
}

@font-face {
    font-family: 'Roboto-Bold';
    src: url('../fonts/roboto-bold-webfont.eot');
    src: url('../fonts/roboto-bold-webfont.eot?#iefix') format('embedded-
opentype'),
         url('../fonts/roboto-bold-webfont.woff2') format('woff2'),
         url('../fonts/roboto-bold-webfont.woff') format('woff'),
         url('../fonts/roboto-bold-webfont.ttf') format('truetype');
    font-weight: normal;
    font-style: normal;
}
```

> **ONDE ENCONTRAR A FONTE ROBOTO?**
>
> A família de fontes Roboto está disponível para download no site oficial da documentação, em https://material.google.com/resources/roboto-noto-fonts.html.

Já no iOS, a família de fontes é a **San Francisco**. Esta é bem interessante e merece algumas explicações.

Acontece que a Apple sempre usou a fonte *Helvetica*, porém, com o surgimento do Apple Watch, ela mostrou-se uma fonte de difícil leitura em telas muito pequenas, por isso foi criada a San Francisco. Esta possui muitas características, sendo uma delas a sua forma compacta. Repare nas diferenças na imagem a seguir:

Figura 3.21: Diferença entre formato normal e compacto

Outro detalhe importante é a diferenciação do tipo *Text* e *Display*, em que as fontes do tipo *Text* são para telas pequenas, e as do tipo *Display* são para maiores.

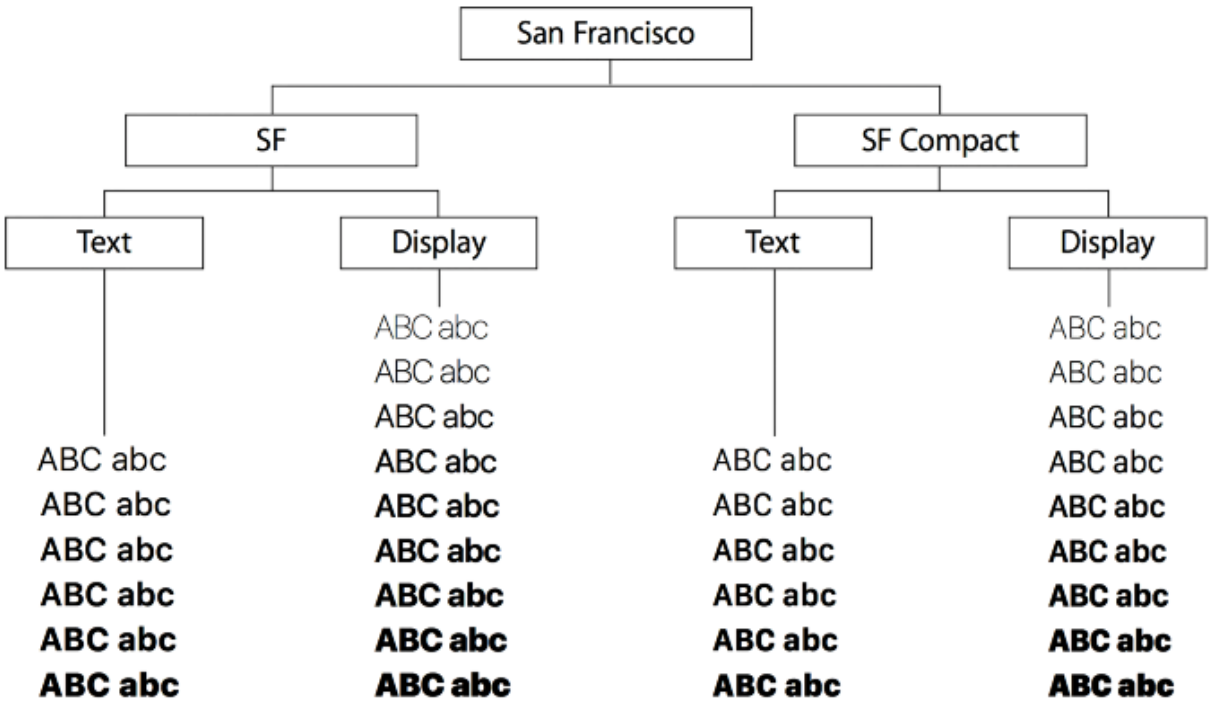


Figura 3.22: Detalhes da fonte San Francisco

E o CSS da fonte ficou desta forma:

```
@font-face {
    font-family: 'SF-UI-Display-Regular';
    src: url('../fonts/sf-ui-display-regular-webfont.eot');
    src: url('../fonts/sf-ui-display-regular-webfont.eot?#iefix')
format('embedded-opentype'),
         url('../fonts/sf-ui-display-regular-webfont.woff2')
format('woff2'),
         url('../fonts/sf-ui-display-regular-webfont.woff')
format('woff'),
         url('../fonts/sf-ui-display-regular-webfont.ttf')
format('truetype');
    font-weight: normal;
    font-style: normal;
}

@font-face {
    font-family: 'SF-UI-Text-Bold';
    src: url('../fonts/sf-ui-text-bold-webfont.eot');
```

```
    src: url('../fonts/sf-ui-text-bold-webfont.eot?#iefix')
format('embedded-opentype'),
         url('../fonts/sf-ui-text-bold-webfont.woff2') format('woff2'),
         url('../fonts/sf-ui-text-bold-webfont.woff') format('woff'),
         url('../fonts/sf-ui-text-bold-webfont.ttf') format('truetype');
    font-weight: normal;
    font-style: normal;
}
```

> **Baixe a fonte San Francisco**
>
> A família de fontes San Francisco está disponível para download no site oficial da documentação https://developer.apple.com/fonts/.

Agora que já conhecemos as famílias de fontes e importamos para o CSS via `@font-face`, vamos criar algumas classes para nos ajudar:

```
.Roboto-Regular{
    font-family: 'Roboto-Regular';
}

.SF-UI-Display-Regular{
    font-family: 'SF-UI-Display-Regular';
}
```

A estratégia será a mesma: carregar uma fonte diferente para uma plataforma. Porém, agora vamos usar uma diretiva do AngularJS muito útil, a `ng-class`. Com esse recurso, podemos adicionar classes seguindo uma condição lógica. Também conseguiremos aplicar uma classe a uma `div` se uma variável tiver determinado valor, jogando essa lógica no `ng-class`. No arquivo `index.html`, logo abaixo da tag `body`, temos uma `div` com `ng-class` que aplica essas classes de fonte, que são baseadas no sistema operacional coletado lá na inicialização:

```
<div ng-view ng-class="{'Roboto-Regular' : device == 'android', 'SF-UI-
Display-Regular' : device == 'ios'}"></div>
```

Com isso, avançamos mais um degrau para entregar um design mais nativo ao usuário final.

3.8 Especificando botões de ação

Normalmente, o que se define em botões são:

- Tipos (`.primary` , `.default` , por exemplo)
- Classes de apoio (para botões sem fundo, por exemplo)
- Customização do estado dos botões (`:hover` , `:active`)

Para o nosso layout, vamos nos atentar aos detalhes do botão `Fechar pedido` .

Figura 3.23: Rodapé da aplicação, onde está o botão "Fechar Pedido"

A tipografia da descrição dos botões é diferente da regular, por isso importaremos também as versões *Bold* das famílias de fontes que conhecemos. A diferenciação será feita usando as classes de apoio `.ios` e `.android` na marcação:

```
.footer-pedido button.android{
    font-family: 'Roboto-Bold';
}

.footer-pedido button.ios{
```

```
    font-family: 'SF-UI-Text-Bold';
}
```

3.9 Escolhendo os ícones

Em aplicativos Cordova, temos a opção de servir ícones em formato fonte com `@font-face`, SVG ou `.png` – o menos indicado por se tratar de uma imagem não escalável e perder a qualidade em determinados momentos.

Os ícones fazem toda a diferença no design do aplicativo. Das opções que temos, o **SVG** e os ícones em formato fonte são as melhores, porque oferecem boa qualidade e são escaláveis.

Os sites que mais utilizo são:

- https://icomoon.io – É possível exportar como SVG, SVG Sprite, PNG e em formato fonte.
- https://www.flaticon.com – É possível exportar com SVG e importar no Icomoon; possui uma grande quantidade de ícones Flat.

Ícones em código SVG

Uma imagem SVG pode ser inserida de diversas formas no código. A forma mais simples é pela tag `<img>`, porém, não é possível manipular esse SVG depois. Escolhi declarar o código do SVG no HTML, em alguns ícones do aplicativo, pois isso nos possibilita manipular a imagem para criar animações no futuro.

Os arquivos em que você encontrará ícones em código SVG serão:

- `index.html`
- `home.html`
- `pedido-busca.html`
- `pedido-info.html`

Ícones em @font-face

Também utilizei um `@font-face` declarado no arquivo `snackbar.css`. Nessa fonte, importo dois ícones que usamos na *Snackbar*, o de sair e o de voltar. Estes escolhi no site https://icomoon.io/app, no pacote *Material Icons*.

Figura 3.24: Exportando ícones no Icomoon

Depois de escolher e exportar o ícone, você terá uma pasta com o CSS já pronto e os arquivos das fontes. Copie e cole o CSS dentro do arquivo `snackbar.css`. Veja a seguir a declaração reduzida do código, apenas para termos uma ideia:

```
@font-face {
  font-family: 'snackbar-icones';
}

.icon-back:before {
  content: "\e5c4";
}

.icon-exit:before {
  content: "\e879";
}
```

Também copie as fontes para `www -> assets -> fonts`. Agora para usar, basta chamar a classe dos ícones:

```
<i class="icon-back"></i>
```

Ícones em SVG Sprite

Outro recurso que vamos usar é o de SVG Sprite. Com o site do *Icomoon*, é possível selecionar vários e baixá-los em formato SVG. O mais interessante é que ele também cria o *Sprite*.

Se você ainda não sabe, um *Sprite* é a união de várias imagens em uma única. Com um Sprite SVG, podemos usar um recurso muito interessante dele, que é a declaração `xlink:href` :

```
<svg>
    <use xlink:href="assets/img/icons.svg#icon-map-ios"></use>
</svg>
```

Perceba que esse código faz referência a uma imagem, chamando pela hashtag `#icon-map-ios` . Essa declaração insere o ícone declarado como um `symbol` na imagem `icons.svg` . Este recurso é bastante útil, pois mantém a legibilidade do código HTML e nos dá uma melhor organização dos ícones SVG da aplicação.

A imagem `icons.svg` encontra-se em `app -> assets -> img` . Ela foi gerada a partir do site do Icomoon com o recurso *Generate SVG*.

> No *capítulo Hooks, armazenamento local, segurança e muito mais!*, vamos trabalhar a animação de dois ícones usando SVG em seu formato puro. Por hora, o que fizemos já é suficiente. Vale lembrar que a Web evolui tão rápido que talvez, quando estiver lendo esta seção, existam outras possibilidades mais rápidas e semânticas de se trabalhar ícones.

3.10 Escolhendo uma paleta de cores

Tanto o Android quanto o iOS seguiram a linha do *Flat Design*, como explicamos no início do capítulo em *Seguindo e entendendo as tendências do design mobile*, portanto, aqui podemos usar a mesma paleta de cores para as duas plataformas, sem problemas. As cores primárias escolhidas serão:

Figura 3.25: Paleta de cores do Freeburguer

OUTRAS VARIAÇÕES DE CORES

Baseado nessas cores da paleta, podemos coletar variações. Um serviço bastante útil para isso é o site http://www.0to255.com. Para saber a variação de uma cor, basta adicionar a cor hexadecimal na URL, como: http://www.0to255.com/212121.

Paletas de cores

A documentação do Google Material Design possui uma página exclusiva com diversas paletas de cores e variações de uma mesma tonalidade. Seu link é http://bit.ly/2heCKhP. Vale a pena conferir!

3.11 Criação de alertas visuais

Na tela do pedido, vamos precisar utilizar dois tipos de alertas: um deles será feito com HTML, CSS e JavaScript; e o outro usando o sistema nativo de alertas com ajuda do plugin `cordova-plugin-dialogs`.

O primeiro alerta será feito usando nossas habilidades de web (HTML, CSS e JS). Escolhi o modelo *Snackbar* – descrito no guia do Material Design, feito pela Google (http://bit.ly/2i78wLS). Como o iOS segue o design Flat, e as cores do *Snackbar* são flat, não vejo problema em adotá-lo também para o iOS. Até o momento, não encontrei uma especificação nos documentos da Apple que ditam um sistema de alertas como o *Snackbar*.

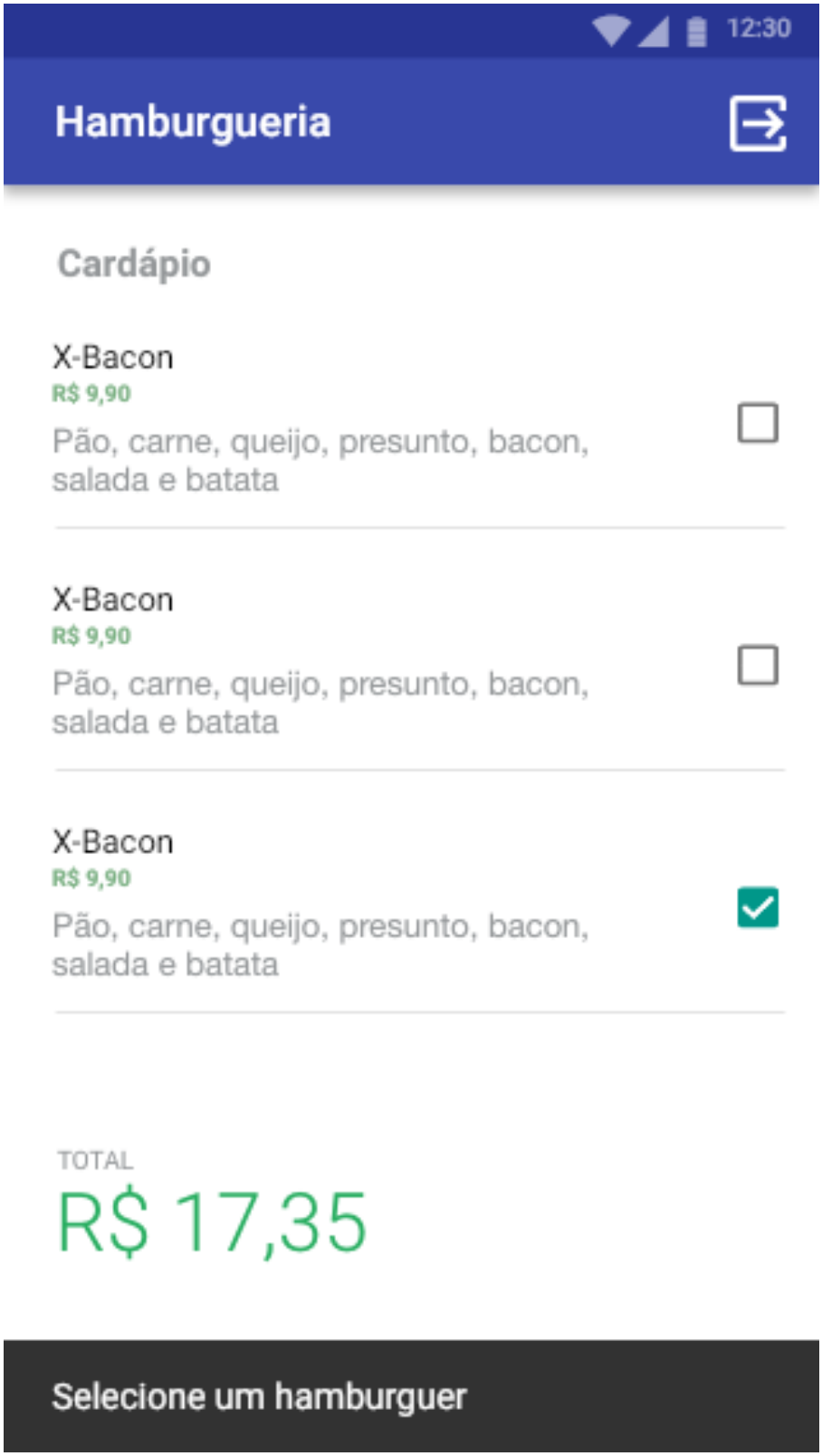


Figura 3.26: Snackbar para alertas simples

Muitas vezes, quando for construir um aplicativo, você vai reutilizar código de componentes já prontos. Isso é uma boa prática na nossa área, e farei isso com o *Snackbar*, usando o código de um projeto meu hospedado no GitHub. Você pode facilmente baixá-lo em http://bit.ly/2hwHBZn.

Baixados os arquivos do projeto, basta adicionar o link do CSS ao topo da nossa aplicação, e o JavaScript ao final, antes do fechamento do `</body>`.

Se o link do projeto *Snackbar* for feito corretamente, vamos ter disponível no JS o objeto `snackbar`. Este possui três métodos:

- `snackbar.show(message)` : exibe o alerta para o usuário;
- `snackbar.hide()` : retira o alerta da tela;
- `snackbar.timer(message, miliseconds)` : programa um tempo para o alerta aparecer e sumir sozinho.

A segunda escolha é o uso do plugin do Cordova, que possibilita utilizar alertas nativos do sistema. Esse plugin possui os seguintes métodos:

- `navigator.notification.alert` : emite um alerta simples;
- `navigator.notification.confirm` : emite um alerta com possibilidade de cancelar ou aceitar;
- `navigator.notification.prompt` : emite um alerta com campo de entrada de dados;
- `navigator.notification.beep` : emite sinais sonoros.

Dentre esses alertas, o que nos interessa neste momento é o `confirm`; a próxima imagem ilustra um exemplo dele. Vale lembrar que, por ser plugin, ele é nativo da plataforma, ou seja, ele terá o layout nativo do *confirm* do iOS.

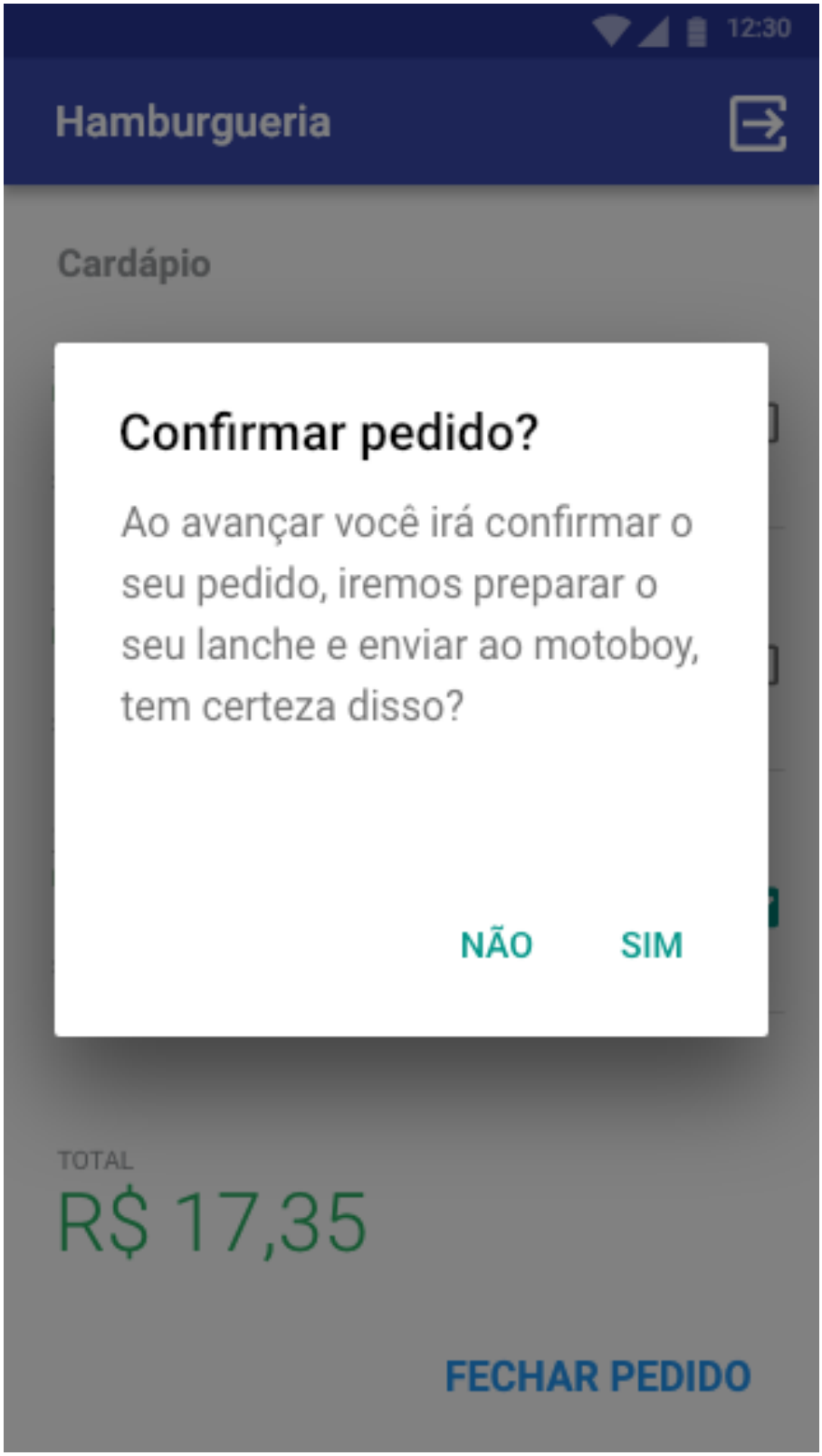


Figura 3.27: Exemplo do resultado que queremos

Para usar o plugin, primeiro vamos precisar adicioná-lo ao projeto. No terminal, acessamos a raiz da aplicação e rodamos o comando a seguir:

```
cordova plugin add cordova-plugin-dialogs --save
```

Para o Freeburguer, esses dois meios de alerta (`snackbar` customizado com HTML+CSS+JS e o `confirm` do plugin `cordova-plugin-dialogs`) serão suficientes.

3.12 Sistema de grid

Cabe a você avaliar a necessidade de usar um grid pronto, ou utilizar os seus conhecimentos em CSS para alinhar os elementos. Um sistema de grid é útil para uma aplicação com layout complexo. Agora, se você deseja apenas alinhar alguns poucos elementos, vale mais a pena usar um `float` reutilizável para a esquerda ou a direita.

É importante lembrar que, quanto mais código desnecessário, mais trabalho o WebView terá para processar o aplicativo.

No caso do Freeburguer, não será necessário um grid completo, por se tratar de um layout muito simples. Porém, você vai se deparar com aplicativos de layout mais complexo. Para isso, recomendo que utilize um framework. Os que mais uso são:

1. Flexboxgrid (http://flexboxgrid.com): fácil de usar, sintaxe parecida com Bootstrap;
2. Ungrid (https://github.com/chrisnager/ungrid): bem leve, fácil de usar;
3. MaterializeCSS (http://materializecss.com/grid.html): sintaxe simples e lembra o Bootstrap.

Se você precisar criar o seu próprio alinhamento de forma mais otimizada, sem ter de carregar um arquivo com muito código

desnecessário, a dica é usar o `Flexbox` ou o `Grid Layout` da especificação do CSS. No nosso caso, escolhi não usar nenhum framework desses de grid, pois o layout do Freeburguer é bem simples.

3.13 Transições e animações

As transições que utilizam animações tornam a interface da aplicação mais fluida e elegante, se usadas com moderação, claro. Com AngularJS, temos duas possibilidades para trabalhar transições: **usando o sistema de rotas**, ou **na própria página**.

Usando o sistema de rotas

No AngularJS, existe um módulo chamado `ngAnimate`, que permite criar transições CSS entre uma rota e outra. Essa diretiva nos ajuda injetando automaticamente classes CSS quando a rota muda. Logo, podemos customizar animações ao utilizar CSS baseado nesses *estados* injetados no elemento.

Dentro do arquivo `freeburguer -> www -> index.html`, vamos olhar para a `div` que está marcada com o parâmetro do AngularJS, chamado `ng-view`. Essa marcação dita ao framework que, a cada troca de rota, as views que estão definidas dentro do arquivo `freeburguer -> www -> app -> bootstrap.js` serão injetadas dentro dessa `div`.

Para utilizar esse método de transição, é necessário:

- `ngRoute`: módulo de roteamento das páginas;
- `ngAnimate`: módulo para criar as classes necessárias para a animação;
- CSS animations: animações que serão aplicadas às páginas.

A imagem a seguir ilustra como funciona a lógica das animações ao trocar de rota:

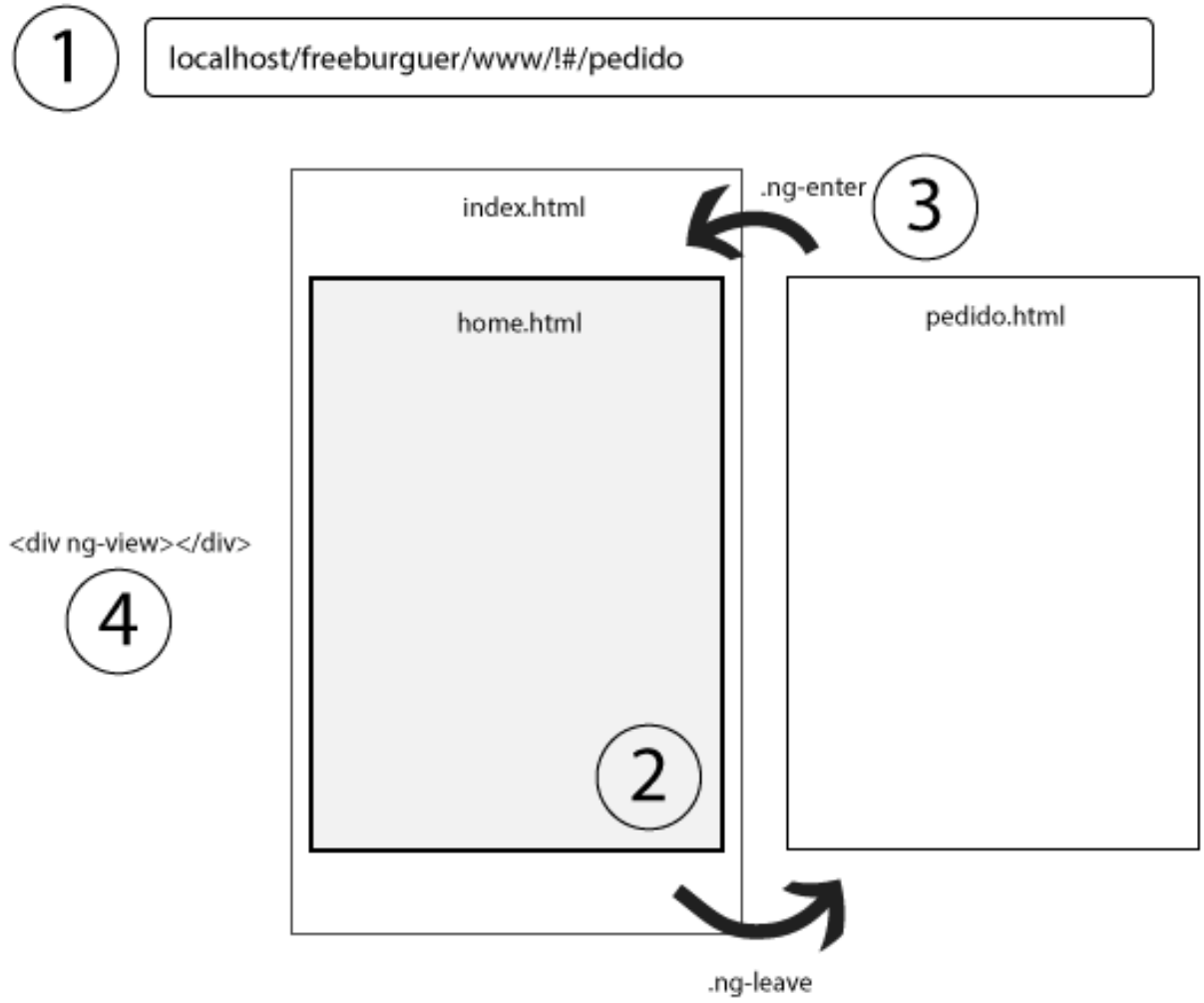


Figura 3.28: Funcionamento do ngAnimate com o sistema de rotas

Explicando os pontos da imagem anterior:

1. Ocorre a troca da rota, seja por um botão, redirecionamento ou mesmo diretamente no navegador.
2. O *pedaço* de view da rota anterior recebe uma injeção de classes: `.ng-leave` e `.ng-leave-active`.
3. A rota nova entra em ação e, com ela, vem o *pedaço* de view nova. Essa view recebe uma injeção de `.ng-enter` e `.ng-enter-active`.
4. Isso tudo ocorre de forma assíncrona na `div` em que está marcada a diretiva que trabalha junto com as rotas, chamada `ng-view`.

Compreendido como funciona o sistema do `ngAnimate`, agora vamos à prática. No CSS, começaremos criando a primeira marcação referente à animação:

```
.home, .slide{
    position: fixed;
    left: 0;
    right: 0;
    bottom: 0;
    top: 0;
}
```

A classe `.home` será usada para customizar exclusivamente a página inicial, e a outra classe chamada `.slide` será para determinar a animação. Como podemos ver, é possível termos várias classes para diferentes animações e comportamentos.

Agora, vamos determinar que acontecerá uma transição CSS toda vez que os *estados* `.ng-enter` e `.ng-leave` ocorrerem nas classes de transição que determinamos anteriormente.

```
.slide.ng-enter,
.slide.ng-leave,
.home.ng-enter,
.home.ng-leave{
    -webkit-transition: all 300ms cubic-bezier(0.4, 0, 0.2, 1);
    transition: all 300ms cubic-bezier(0.4, 0, 0.2, 1);
}
```

Podemos escolher o que vai acontecer quando ocorrer uma entrada da view na aplicação. Primeiro, olhe para a imagem a seguir para entender por que o `ngAnimate` injeta duas classes – por exemplo, `.ng-enter` e `.ng-enter-active`:

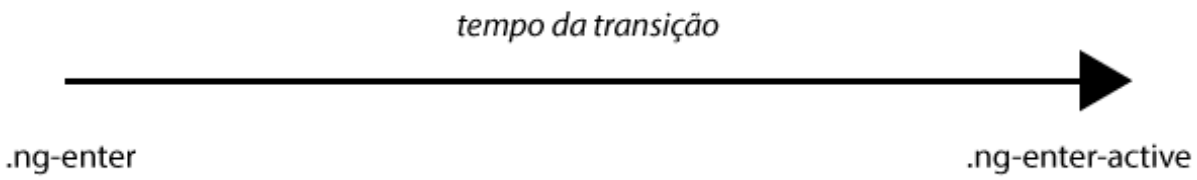


Figura 3.29: Como devemos pensar sobre as classes de estado

```
.slide.ng-enter {
    transform: translateX(100%);
}

.slide.ng-enter-active {
    transform: translateX(0%);
}
```

> Repare que a transição será executada em um tempo de **300ms** (milissegundos), utilizando uma função de tempo `cubic-bezier(0.4, 0, 0.2, 1)`. O Google possui uma página especial que aborda como trabalhar com o tempo ideal e o movimento natural de elementos. Vale a pena ler com calma: http://bit.ly/2hfRNcm.

A mesma lógica serve para o estado de saída da view:

```
.slide.ng-leave,
.home.ng-leave {
    transform: translateX(0%);
    opacity: 1;
}

.slide.ng-leave-active,
.home.ng-leave-active {
    transform: translateX(-100%);
    opacity: 0;
}
```

Por fim, queremos que a primeira view não seja animada. Não faz sentido algum fazer uma transição logo que entramos no aplicativo, por isso, vamos criar a classe `.home`, para nos ajudar. Basta não

fazer nenhuma transição quando estiver em estado de entrada `.ng-enter`:

```
.home.ng-enter {
    transform: translateX(0%);
}

.home.ng-enter-active {
    transform: translateX(0%);
}
```

Para finalizar, como estamos trabalhando duas classes de transição, a `.home` e a `.slide`, precisamos saber a hora de adicioná-las e retirá-las. Com AngularJS, isso é bem fácil de fazer. Dentro do arquivo `bootstrap.js`, essa função `.run` é executada logo depois de carregar o AngularJS, e é ideal para fazer configurações iniciais.

Com isso, podemos instalar uma *escuta*. Toda vez que acontecer a troca de rota na aplicação, capturamos a URL atual. Logo, nosso trecho de código ficou assim:

```
$rootScope.$on("$locationChangeStart", function (event, next, current) {
    $rootScope.path = $location.path();
});
```

O `$rootScope` é uma variável global dentro do AngularJS, por isso, podemos acessar o `path` em qualquer lugar, independente de rota. Logo, vamos criar mais duas condicionais, assim como fizemos com as tipografias, para determinar quando adicionar as classes com base no valor da variável `path`:

```
<div ng-view ng-class="{home: path === '/', slide: path !== '/', 'Roboto-Regular' : device == 'android', 'SF-UI-Display-Regular' : device == 'ios'}"></div>
```

Finalizamos a configuração básica de transições com rota. Você pode explorar bem essas animações para tornar a aplicação mais fluida, tendo um certo cuidado com exageros.

Inspirações

Você pode encontrar algumas inspirações do que pode ser feito com transições em CSS no link http://bit.ly/2ieJzBC.

Transições CSS para animar elementos do aplicativo

Primeiro, falamos do uso de transições CSS em união do sistema de rotas do AngularJS para criar aqueles efeitos suaves de transição ao passar de uma página para outra. Agora falaremos de transições dentro da própria página HTML.

Isso serve para dar suavidade aos elementos na página. Você pode tirar um elemento da página de forma *seca*, por exemplo, sem nenhuma animação; ou pode aplicar um efeito que tornará a exibição mais agradável aos olhos do usuário. A seguir, vamos animar a área onde o aplicativo exibe o total do pedido.

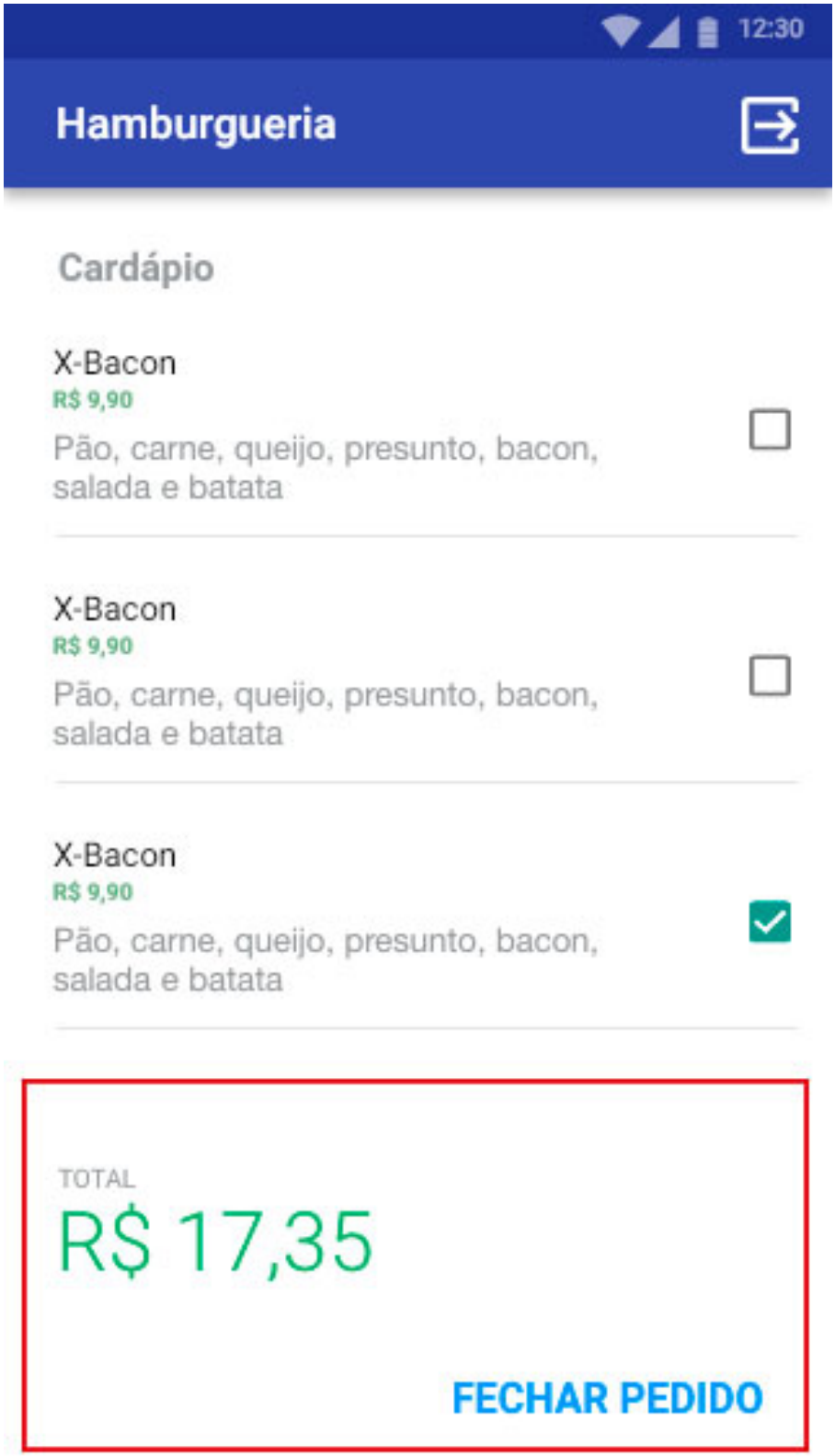


Figura 3.30: Local onde queremos animar

A animação que vamos trabalhar é uma `div` que exibe as informações totais do pedido. A ideia é fazer com que ela apareça de forma suave na tela, e não de maneira bruta. A seguir, confira seu esquema visual:

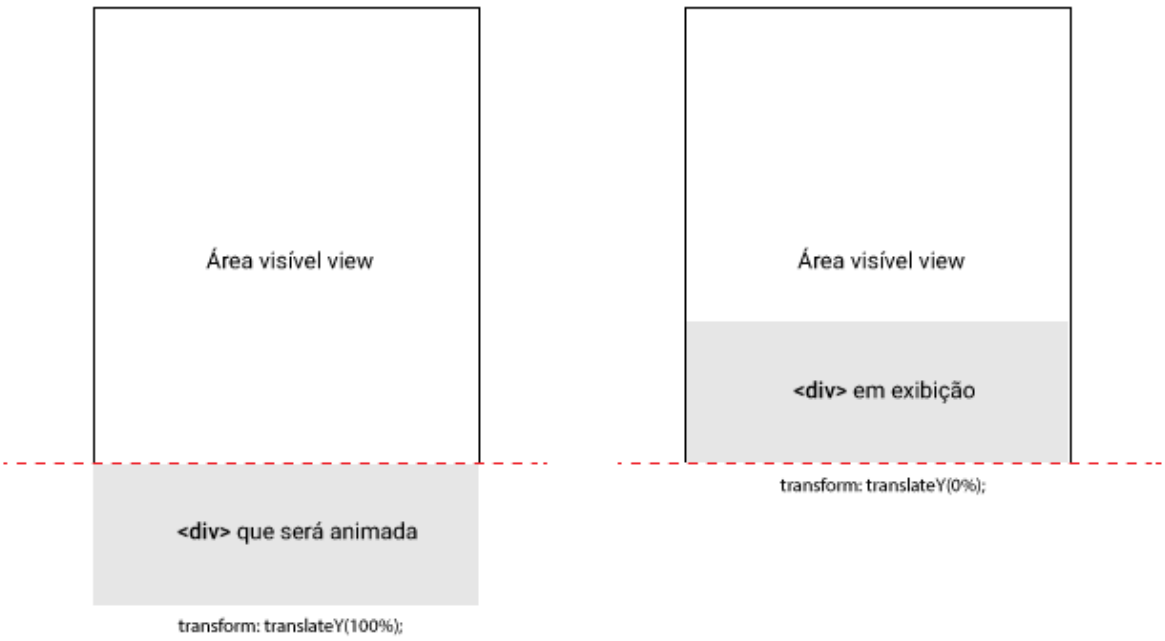


Figura 3.31: Ideia da transição do componente

Vamos criar duas classes de animação, sendo que cada uma vai executar uma animação diferente:

```
.abrir-footer,
.fechar-footer{
    animation-duration: 300ms;
    animation-timing-function: ease;
    animation-fill-mode: both;
}
.abrir-footer{
    animation-name: abrirFooter;
}

.fechar-footer{
    animation-name: fecharFooter;
}
```

A animação `abrirFooter` ficou com o seguinte CSS:

```
@keyframes abrirFooter {
    from {
        transform: translateY(100%);
    }
    to {
        transform: translateY(0%);
    }
}
```

Obviamente, a `fecharFooter` faz o contrário da anterior:

```
@keyframes abrirFooter {
    from {
        transform: translateY(0%);
    }
    to {
        transform: translateY(100%);
    }
}
```

Podemos usar um recurso do AngularJS chamado `$watch`, que fica observando a mudança de valores da variável `itens`. Sempre que tiver algum item da lista do *array* selecionado, somamos o valor total e marcamos a variável de apoio (`exibirResumoPedido`) como `true`. Sabendo disso, vamos usar essa mesma variável para determinar as animações de abrir e fechar o elemento.

Neste capítulo, ainda não criamos os controladores da aplicação AngularJS, mas a seguir você confere o que o `$watch` fará futuramente. Repare principalmente na variável `$scope.exibirResumoPedido`:

```
$scope.$watch('itens', function(){

    var selecionados = 0;
    var itens = [];
    $scope.valorTotalPedido = 0;
```

```
    angular.forEach($scope.itens, function(item, key){
        if (item.selecionado){
            itens.push(key);
            selecionados++;
            $scope.valorTotalPedido += item.preco;
        }
    });

    $scope.exibirResumoPedido = (selecionados > 0) ? true : false;

}, true);
```

```
<footer class="footer-pedido" ng-show="exibirResumoPedido" ng-class="
{'abrir-footer': exibirResumoPedido, 'fechar-footer':
!exibirResumoPedido}">
```

Repare que o resumo será exibido apenas se `exibirResumoPedido` for verdadeiro. Logo, a classe `.abrir-footer` será injetada no elemento. Ao contrário disso, se `exibirResumoPedido` for falso, a classe `.fechar-footer` será injetada, aplicando ao elemento as animações antes criadas por nós.

> Você pode baixar o aplicativo pronto até aqui em http://bit.ly/cordova-livro. O projeto é o **freeburguer-cap3**.

3.14 Revisão

Chegamos ao final de mais uma etapa. Nosso objetivo neste capítulo foi, acima de tudo, mostrar que é totalmente possível entregar um aplicativo híbrido, com uma interface muito parecida com a nativa, com as técnicas que temos hoje no universo web.

Vimos como é importante se atentar aos mínimos detalhes, como adotar a família de fontes nativas de cada plataforma, e que esses detalhes fazem a diferença para o usuário – mesmo que de forma

inconsciente. Em relação à sua experiência e usabilidade no aplicativo, esses são dois pontos importantes para engajar e motivar as pessoas a voltarem a usar o app.

No próximo capítulo, partiremos para a construção de todas as funcionalidades do aplicativo. Veremos mais a fundo o que o PhoneGap e o Cordova nos possibilitam no nível de integração nativa, e também falaremos de um serviço do Google bastante interessante, chamado *Firebase*. Você não pode perder!

Capítulo 4
Desenvolvendo as funcionalidades do aplicativo

Chegou a hora de desenvolvermos as funcionalidades do aplicativo. Para alimentar uma empresa de hambúrgueres, o cliente nos solicitou um painel web simples para configuração dos lanches. Você pode configurar uma empresa teste acessando o link: https://freeburguer-9708a.firebaseapp.com/#!/.

4.1 Controladores do aplicativo com AngularJS

Vamos separar as responsabilidades do aplicativo com o uso de controladores. Os arquivos estão no diretório `freeburguer -> www -> app -> controllers`:

- `home.js`
- `cardapio.js`
- `pedidoBusca.js`
- `pedidoInfo.js`

Cada controlador terá a anatomia semelhante ao código mostrado a seguir. Seguimos o formato proposto no *Angular 1 Style Guide*, feito pelo importante entusiasta AngularJS, John Papa:

```
(function() {
    'use strict';

    angular
        .module('app')
        .controller('HomeController', HomeController);

    HomeController.$injector = ['$scope'];

    function HomeController($scope){
```

```
    }

})();
```

> **Guia de boas práticas**
>
> John Papa é um desenvolvedor-autor de um dos guias de estilo mais populares de como escrever em AngularJS. Você pode conferir o guia em http://bit.ly/2q982Yz.

Todos os arquivos JavaScript serão isolados com uma função anônima autoexecutável. Ela garante a privacidade das variáveis e das funções ao isolar o código do escopo global. Isso ajuda-nos a proteger de futuros conflitos.

Esse padrão é conhecido como IIFE (*Immediately Invokable Function Expressions*). Para usá-lo, precisamos sempre abrir o código definindo uma função anônima:

```
(function() {
```

Em seguida, na última linha do arquivo, fechamos a função e adicionamos parênteses antes do ponto e vírgula, que vai dizer ao interpretador para "executar" imediatamente a função. Como não retornamos nada, a única coisa que acontecerá é a leitura de todo o código escrito dentro:

```
})();
```

Definimos o modo `strict`, que nos ajuda a diminuir a chance de cometermos alguns erros causados por práticas não recomendadas no JavaScript.

As próximas linhas – mais especificamente em `.controller` – definem o nome do controlador e a função responsável por ele, que no nosso caso terá o mesmo nome do controlador por convenção do estilo que estamos seguindo.

```
angular
.module('app')
.controller('HomeController', HomeController);
```

Tudo no AngularJS precisa ser injetado. Isso faz parte da injeção de dependências do framework e, no controlador, não é diferente. Deixamos isso implícito na linha:

```
HomeController.$injector = ['$scope'];
```

Repare que, no exemplo, só injetamos o `$scope` ; outras dependências como o `$rootScope` também serão injetadas ali.

> **Qual a diferença do $scope e do $rootScope no AngularJS?**
>
> Enquanto o $scope é acessível somente no controlador que você está, o $rootScope é acessível em toda a aplicação.

Por fim, a nossa função principal `HomeController` recebe como parâmetro todos os módulos injetados no `$injector` :

```
function HomeController($scope){}
```

Caso você esqueça de injetar qualquer dependência que esteja usando dentro do controlador, o AngularJS vai disparar um `ReferenceError` no console, por exemplo: `$rootScope is not defined` .

Criamos os controladores seguindo o padrão que escolhemos, portanto, temos os seguintes controladores:

Nome do arquivo	Nome do controlador
home.js	HomeController
cardapio.js	CardapioController
pedidoBusca.js	PedidoBuscaController
pedidoInfo.js	PedidoInfoController

Definindo a responsabilidade de cada rota com um controlador

Um dos motivos de usarmos um framework SPA é poder utilizar o seu **sistema de rotas**. Com o AngularJS, usaremos a dependência `$routeProvider` para dizer as responsabilidades de cada uma.

Dentro das rotas definidas em `bootstrap.js`, usamos agora os nossos controladores para responderem por cada uma delas.

```
$routeProvider
.when('/', {
    templateUrl  : 'app/Views/home.html',
    controller   : 'HomeController',
    controllerAs : 'Home'
})
.when('/cardapio/:id_empresa', {
    templateUrl  : 'app/Views/cardapio.html',
    controller   : 'CardapioController',
    controllerAs : 'Cardapio'
})
.when('/buscar-pedido', {
    templateUrl  : 'app/Views/pedido-busca.html',
    controller   : 'PedidoBuscaController',
    controllerAs : 'PedidoBusca'
})
.when('/pedido-info/:id_pedido', {
    templateUrl  : 'app/Views/pedido-info.html',
    controller   : 'PedidoInfoController',
    controllerAs : 'PedidoInfo'
})
```

Repare que usamos `controllerAs`. Isso cria um **alias** do nome do controlador, assim, dentro da View HTML por quem ele é responsável, podemos chamar funções usando o alias. Vale lembrar, claro, que esse formato só funciona se utilizarmos `this` para definir a função.

Imagine que temos um controlador chamado `PedidoInfoController`, com a seguinte função:

```
this.criar = function(){
    console.log('Cria algo...');
}
```

Agora, poderíamos chamar a função pelos alias dentro da View por quem ele é responsável:

```
<button ng-click="PedidoInfo.criar()">Criar</button>
```

Um último detalhe sobre as rotas do AngularJS é a possibilidade de passarmos parâmetros. Repare que, em `/cardapio` e `/pedido-info`, foi passado um parâmetro. Isso é útil, pois podemos coletar esses valores dinamicamente dentro do controlador usando o objeto `$routeParams`. Como no exemplo `/cardapio/200213`, o valor `200213` faz parte do parâmetro da URL `:id_empresa`, logo, podemos coletá-lo no controlador. Veja:

```
var id_empresa = $routeParams.id_empresa;
```

Adicionando o Firebase ao projeto

Firebase é uma plataforma do Google integrada a várias ferramentas que auxiliam no desenvolvimento de aplicativos. O SDK suporta as linguagens Swift, Objective-C, Java, C++, Unity e, claro, o nosso JavaScript.

Dentre as ferramentas que a plataforma oferece, usaremos a ferramenta de banco de dados (Realtime Database). A grande vantagem é que ele funciona em tempo real, ou seja, não há necessidade de o usuário tocar em nenhum botão para atualizar, e ainda poupa um tempo considerável que teríamos de tirar para desenvolver uma API, usando PHP e MySQL para ler e armazenar os nossos dados, por exemplo.

O banco de dados do Firebase é no formato NoSQL. Ou seja, a forma de armazená-los é pela notação JSON – outra vantagem para nós que estamos escrevendo um aplicativo todo com JS.

O Firebase é uma plataforma que oferece uma série de planos. Um deles é o *Spark*, que é gratuito. Para conhecer todos os planos e seus recursos, acesse https://firebase.google.com/pricing. No nosso caso, usaremos o plano Spark.

Agora que já conhecemos o Firebase, precisamos adicionar o SDK JavaScript em nossa aplicação. Para isso, vamos à documentação da plataforma, em `Adicionar o Firebase ao seu projeto do JavaScript`. Lá você encontrará uma explicação de como carregar a biblioteca toda, ou somente alguns recursos. Lembrando que vamos acessar o link do *script* e salvar dentro da aplicação, em `freeburguer -> www -> assets -> js`, para não dependermos da internet para baixar a biblioteca.

Adicionando o suporte a pt-BR com angular-i18n

Precisamos adicionar ao projeto o arquivo de tradução para `pt-BR` do framework AngularJS. O pacote é o `angular-i18n`:

```
bower install angular-i18n --save
```

Assim, podemos usar filtros com o framework, como o `currency`, para formatar um valor para a moeda brasileira, ou imprimir um valor `float` de `20.53` (armazenado na variável `valorTotal`). Podemos fazer isso como no código a seguir:

```
{{ valorTotal | currency }}
```

Verificando o carregamento dos arquivos JavaScript

O arquivo `freeburguer -> www -> index.html` deve conter todos os arquivos JavaScript que adicionamos ao projeto:

```
<script src="cordova.js"></script>
<script src="assets/js/angular.js"></script>
<script src="assets/js/firebase.js"></script>
<script>
  var config = {
    apiKey: "AIzaSyB1hfmmcipVzXuvisLLY96xV61SeLVnsy8",
```

```
    authDomain: "freeburguer-9708a.firebaseapp.com",
    databaseURL: "https://freeburguer-9708a.firebaseio.com",
    storageBucket: "freeburguer-9708a.appspot.com",
    messagingSenderId: "492191314801"
  };
  firebase.initializeApp(config);
</script>
<script src="assets/js/angular-locale_pt-br.js"></script>
<script src="assets/js/angular-route.js"></script>
<script src="assets/js/angular-animate.js"></script>
<script src="assets/js/snackbar.js"></script>
<script src="app/bootstrap.js"></script>
<script src="app/controllers/home.js"></script>
<script src="app/controllers/cardapio.js"></script>
<script src="app/controllers/pedidoBusca.js"></script>
<script src="app/controllers/pedidoInfo.js"></script>
<script src="app/directives/barFixed.js"></script>
<script src="app/services/firebaseTool.js"></script>
```

Repare que carregamos os arquivos seguindo uma ordem de prioridade, assim, a WebView vai carregar:

1. O framework Cordova;
2. O framework AngularJS;
3. O SDK do Firebase;
4. As configurações do projeto Firebase;
5. A biblioteca `angular-i18n` para a tradução de moedas;
6. A biblioteca `angular-route` para o sistema de rotas;
7. A biblioteca `angular-animate` para a manipulação das transições de rota;
8. O arquivo de configuração do projeto que chamamos de `bootstrap.js`;
9. E os demais arquivos de controllers, services, directives e outros, como o `snackbar.js`.

Verificando se está tudo injetado ao iniciar o AngularJS

Agora que adicionamos as dependências e carregamos no arquivo `index.html`, precisamos injetá-las no projeto AngularJS inicializado

em `freeburguer -> www -> bootstrap.js` . Cada módulo no AngularJS possui um nome único, e é injetado no projeto dentro dos parênteses:

```
angular.module('app',['ngRoute', 'ngAnimate'])
```

4.2 Loading spinner com CSS e AngularJS

Precisamos de um componente para ser exibido enquanto carregamos os dados armazenados no Firebase, ou quando for necessária uma ação para gravar dados ou redirecionar o usuário. É bastante fácil resolver isso, a ideia será:

- Usaremos CSS e animações para criar o elemento.
- No HTML, com a ajuda do AngularJS, definimos um `ng-show` para termos controle da exibição. Se você não sabe, o `ng-show` é um recurso do AngularJS que pode ser adicionado a qualquer elemento HTML da sua página quando somado a uma lógica. No nosso caso, será apenas uma variável `true` ou `false` para dizer se mostrará ou não elemento.
- Por fim, usaremos uma variável global do AngularJS, o `$rootScope` , chamando a variável de `$rootScope.carregar` .

Poderíamos criar do zero esse carregamento com CSS, mas vamos adiantar o trabalho com o site http://tobiasahlin.com/spinkit. Lá temos vários *Spinners* prontos, e vamos optar pela quinta opção. Você pode escolher qual quiser; a seguir, basta clicar em *source* para ver o código CSS e HTML.

No arquivo `freeburguer -> www -> app -> index.html` , logo após a abertura do `<body>` , vamos colar o HTML:

```
<div class="spinner" ng-show="carregar">
  <div class="double-bounce1"></div>
  <div class="double-bounce2"></div>
</div>
```

Então, adicionamos o CSS ao arquivo `freeburguer -> www -> css -> app.css`. Você pode pegar seu código completo em: http://bit.ly/2BvMgUx.

Agora, com a ajuda do AngularJS, toda vez que quisermos ativar o carregamento, basta atribuirmos a variável global como `true`, como o código seguinte:

```
$rootScope.carregar = true;
```

Para desativar, atribuímos a variável como `false`.

4.3 A função para encontrar empresas

Criamos a função JavaScript chamada `encontrar()` dentro do controlador `HomeController` com o objetivo de buscar uma empresa pelo seu código. Esse código é passado por parâmetro para a função `encontrar(codigo)`.

O **código** é uma identificação única da empresa, gerado aleatoriamente na hora do cadastro no site, com a letra `F` e cinco dígitos numéricos (por exemplo, *F39133*). Uma das coisas que a função `encontrar(codigo)` faz é buscar no Firebase, portanto, precisamos inicializar uma instância do banco:

```
var db = firebase.database();
```

Vale lembrar que, no código anterior, a variável `firebase` é uma variável vinda do código SDK (*Software Development Kit*) JavaScript definido no arquivo `firebase.js`, que importamos no `index.html`.

O próximo passo é referenciar a área na árvore JSON do banco que queremos buscar. Lembra que o Firebase usa uma base no formato NoSQL? Então, todo banco de dados é formato por ramificações de objetos JSON. Nossos dados estão estruturados em duas grandes ramificações, empresas e pedidos:

- freeburguer-9708a (nome da base)
  - **empresas**
    - empresa 1
    - empresa 2
    - ...
  - **pedidos**
    - pedido 1
    - pedido 2
    - ...

Para buscar uma empresa, precisamos criar uma referência para a ramificação `empresas`. Para isso, faremos:

```
var empresas = db.ref('empresas');
```

Agora vamos rodar uma espécie de SQL dentro do Firebase. O código a seguir vai tentar buscar, dentro das empresas, uma linha que contenha o código igual ao passado na função:

```
var query = empresas
                .orderByChild('codigo')
                .equalTo(codigo)
                .limitToFirst(1);
```

Regra de indexação

Como estamos realizando buscas, o Firebase vai mandar um `warn` avisando para que façamos uma indexação do atributo código. Para isso, dentro das *regras* do banco de dados no painel do Firebase, adicionamos:

Figura 4.1: Menu no Firebase, dentro de Database

```
"empresas": { ".indexOn": ["codigo"] },
```

Com a variável `query`, verificamos **se** foi encontrada. Em caso positivo, redirecionamos para a página da empresa; **caso contrário**, vamos vibrar o aparelho por 100 milissegundos e exibir um alerta nativo ao usuário:

```
query.on('value', function(snapshot){

    if (snapshot.val()){
        // Achou
    }else{
        // Não achou
    }
});
```

Para fazer *vibrar*, vamos usar um plugin. Acesse o diretório raiz do projeto e adicione:

```
cordova plugin add cordova-plugin-vibration --save
```

Quer entender como funcionam as consultas no Firebase?

A documentação oficial da plataforma (http://bit.ly/2h7GCnq) e o post *Firebase Database para Desenvolvedores SQL — Queries simples* (http://bit.ly/2zghX6P), do Rosário Fernandes, podem ajudá-lo a entender melhor o sistema de buscas.

4.4 A leitura do QRCode da empresa

O código para ler o QRCode será feito no controller `home.js`. Precisamos injetar algumas dependências do AngularJS que vamos usar no código:

```
HomeController.$injector = ['$scope', '$rootScope', '$location',
'$timeout'];
```

O `$scope` é a variável no nível do controller; `$rootScope` é a variável global; `$location` é para manipulação da URL e redirecionamentos; e `$timeout` é uma diretiva referente à função `setTimeout()` do JavaScript.

O primeiro passo será adicionar o plugin. Vamos ao terminal até o diretório raiz do aplicativo, para rodarmos o seguinte comando:

```
cordova plugin add phonegap-plugin-barcodescanner --save
```

Feito isso, agora estamos prontos para usar o objeto injetado. Seu código será simples:

```
cordova.plugins.barcodeScanner.scan(
    function (result) {
           console.log("O código do QRCode é:" + result.text);
    },
      function (error) {
        console.error("Erro ao scanear qrcode: " + error);
```

```
    }
);
```

A sintaxe desse plugin é bem direta. Assim, vamos dar um direcionamento ao *código da empresa* encontrado no QRCode. Se cairmos na função de sucesso, chamamos a função `encontrar(codigo)`, que vai efetuar a busca no Firebase:

```
cordova.plugins.barcodeScanner.scan(
    function (result) {
           encontrar(result.text);
    },
      function (error) {
        console.error("Erro ao scanear qrcode: " + error);
    }
);
```

Agora que sabemos como usar o plugin, vamos acoplar a funcionalidade ao toque do botão visual do QRCode da tela inicial. Usaremos o `ng-click` do AngularJS para chamar a função `scanear()` que vai executar o plugin.

Na View `home.html`, temos o botão definido:

```
<button ng-click="Home.scanear()" ng-show="!codigo_empresa">
    <i class="qrcode color-button-home"></i>
</button>
```

Portanto, ao tocar no botão, será executada a função `scanear()`, definida no controller `home.js`.

4.5 Uma busca direta pelo código da empresa

Uma outra opção é quando o usuário tocar no `input` e inserir o **código** da empresa diretamente. Para isso, vamos sumir com o `button` e apresentar um outro `button`, que executará a função `buscar()`. Primeiro, vamos olhar para o input:

```
<input type="text" ng-model="codigo_empresa" placeholder="Código da casa"
maxlength="6">
```

O parâmetro `ng-model` é do AngularJS. Ele coleta tudo o que é digitado no `input` para dentro de `$scope`. Fica fácil passarmos o código para a função `encontrar()`, assim como fizemos com o caso do QRCode:

```
this.buscar = function(){
    encontrar($scope.codigo_empresa);
}
```

No código a seguir, vemos que o botão (com descrição) aparecerá caso a variável `codigo_empresa` exista. Já o outro botão (com ícone) de negação só vai aparecer quando ela não existir.

```
<button ng-show="codigo_empresa">buscar</button>

<button ng-show="!codigo_empresa">
    <i class="qrcode color-button-home"></i>
</button>
```

Hora de testar!

Agora que estamos produzindo código, pode ocorrer algum erro que não será possível identificar sem vermos o aplicativo rodando. Isso acontece pois já sabemos que o Cordova e os plugins que instalamos só estarão disponíveis depois de injetados, algo que só ocorre em execução no aparelho.

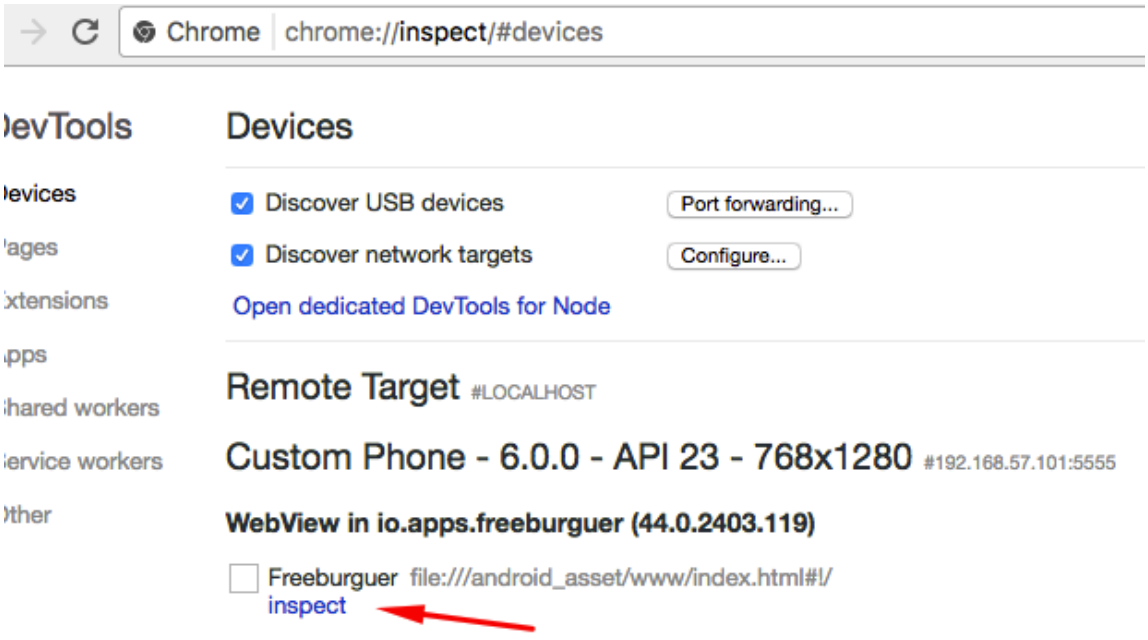
Para isso, usaremos um recurso de **inspecionar** o projeto em execução. A seguir, explico como fazer isso no Android e, na sequência, no iOS.

4.6 Android: como usar o Chrome para inspecionar o projeto em execução

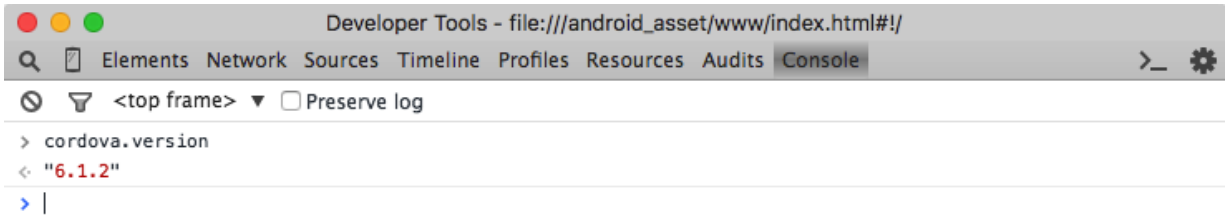
Você pode inspecionar um aparelho no *simulador*, ou conectar um aparelho real via USB. Em ambos os casos, siga as configurações:

> Se você estiver usando Windows, por favor verifique se os drivers estão instalados: https://developer.android.com/studio/run/oem-usb.html.

1. No Android, vá em `Configurações -> Opções de desenvolvedor -> Ativar depuração USB`.
2. Na sua máquina, abra o Google Chrome pela URL `chrome://inspect`.
3. Certifique-se de que a opção `Descobrir dispositivos USB` está selecionada.
4. Conecte o aparelho via USB (não use hubs). Na primeira vez que conectar, o Android solicitará uma autorização sua; basta aceitar a depuração.
5. Agora, na página de inspeção `chrome://inspect`, você verá o dispositivo conectado ao aplicativo. Basta clicar em `inspect` para abrir o terminal.



Feito isso, você agora tem um terminal dedicado para fazer verificações de erros, ou mesmo testar algum plugin, já que agora é possível testar plugins injetados usando o *console* e visualizando no aparelho sua execução:

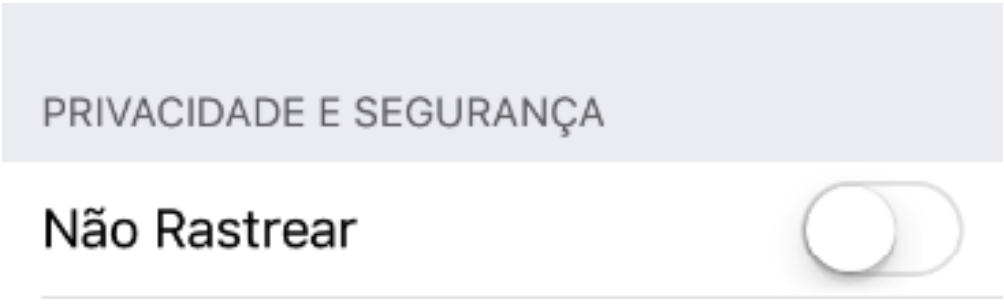


Além do console, podemos também mudar o CSS e remover elementos, tudo em tempo real. Em alguns momentos do desenvolvimento, isso pode ser bem útil.

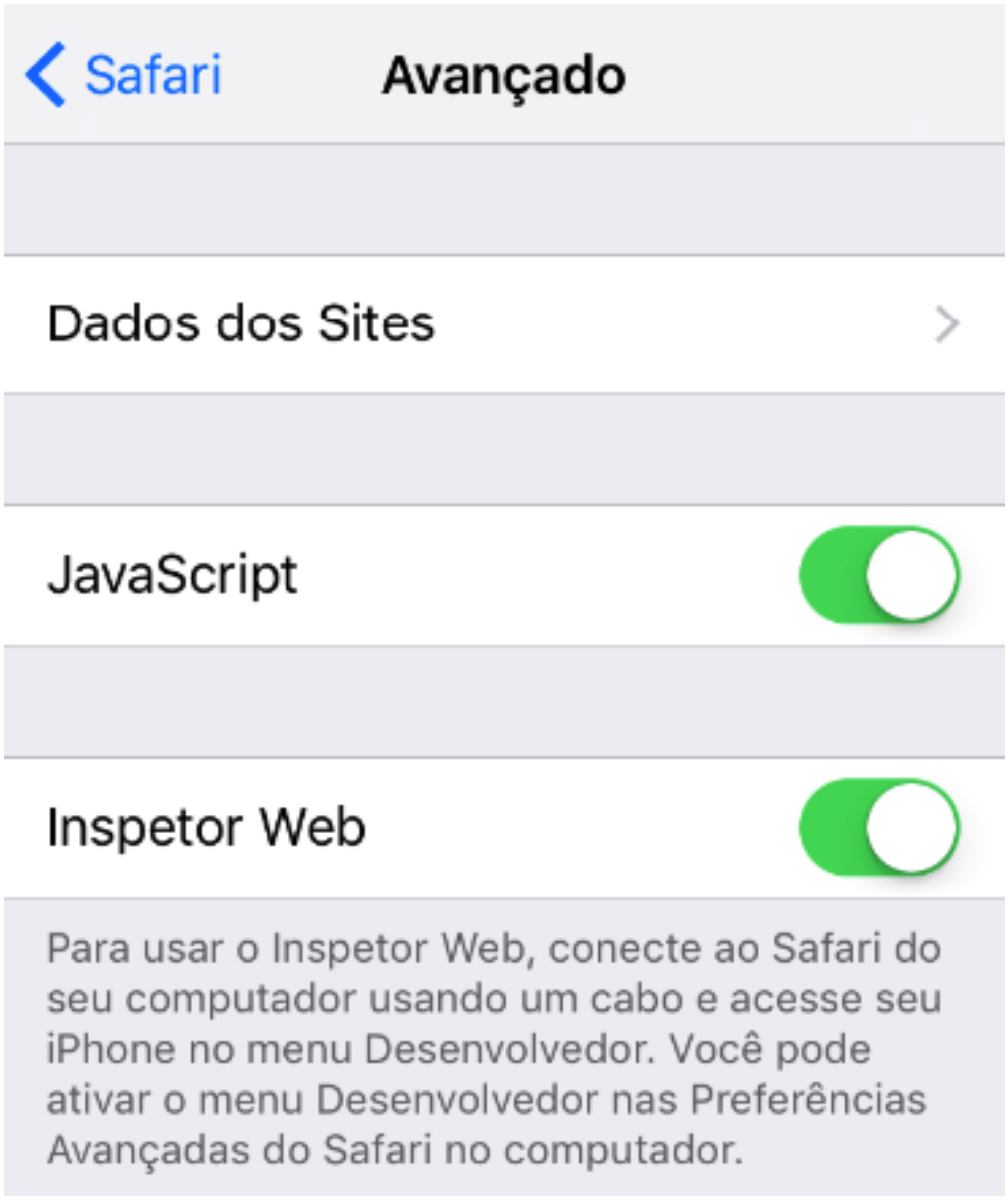
4.7 iOS: como usar o Safari para inspecionar o projeto em execução

Você pode inspecionar um aparelho no *simulador* ou conectar um aparelho real via USB. Em ambos os casos, siga as configurações:

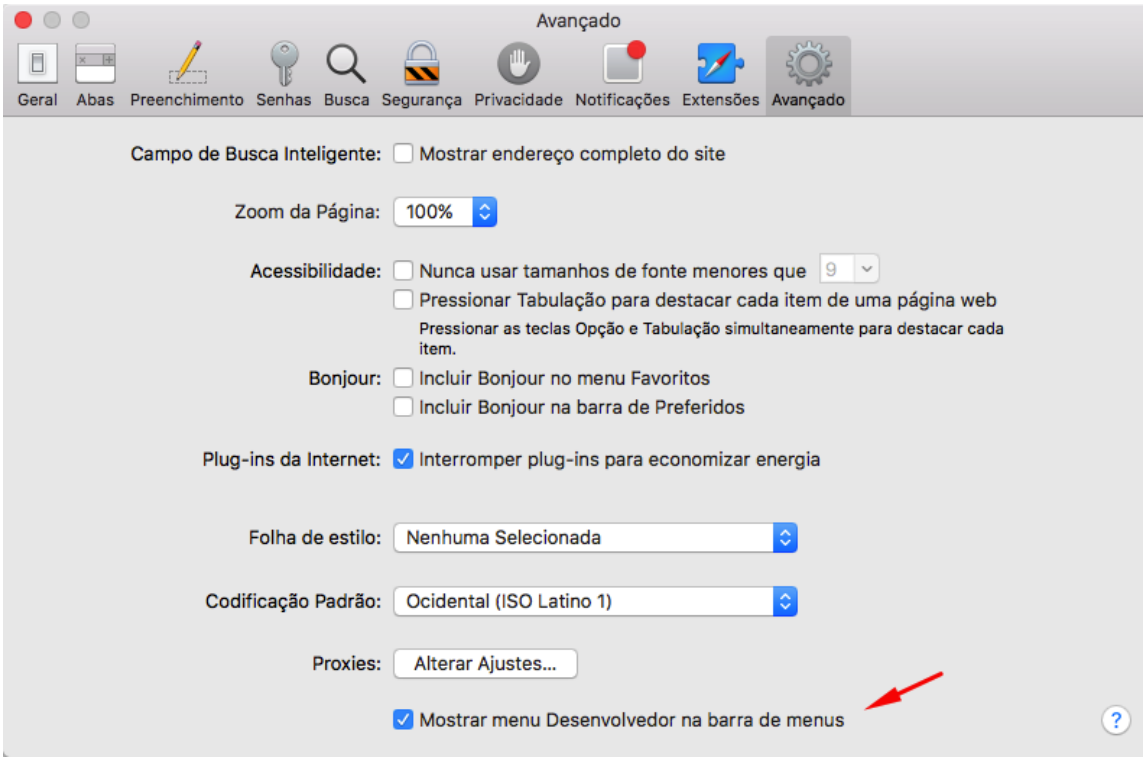
1. No aparelho móvel (iPhone/iPad), vá em `Ajustes -> Safari` , e desligue a opção `Não Rastrear` .



2. Ainda nessa tela, vá em `Avançado` e ative a opção `Inspetor Web` .

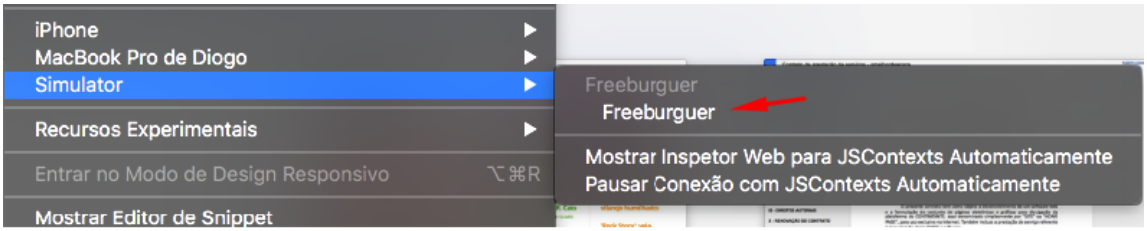

Safari
Avançado
Dados dos Sites
JavaScript
Inspetor Web
Para usar o Inspetor Web, conecte ao Safari do seu computador usando um cabo e acesse seu iPhone no menu Desenvolvedor. Você pode ativar o menu Desenvolvedor nas Preferências Avançadas do Safari no computador.

3. Agora no MacOS, abra o Safari, vá em `Preferencias -> Avançado` e ative a opção `Mostrar menu Desenvolvedor na barra de menus .`



4. Inicialize o aplicativo usando o Xcode no modo simulador, ou rodando-o em seu aparelho físico (nesse caso, conecte o aparelho via USB).

5. Depois de inicializar o app, vá até o Safari em seu desktop, e abra o menu `Desenvolvedor -> iPhone -> Freeburguer`.



Assim, você terá um terminal dedicado para fazer verificações. Por exemplo, caso um plugin falhe, você poderá ir até o menu `console`, como também usar o console para testar plugins injetados no projeto; tudo em tempo real.

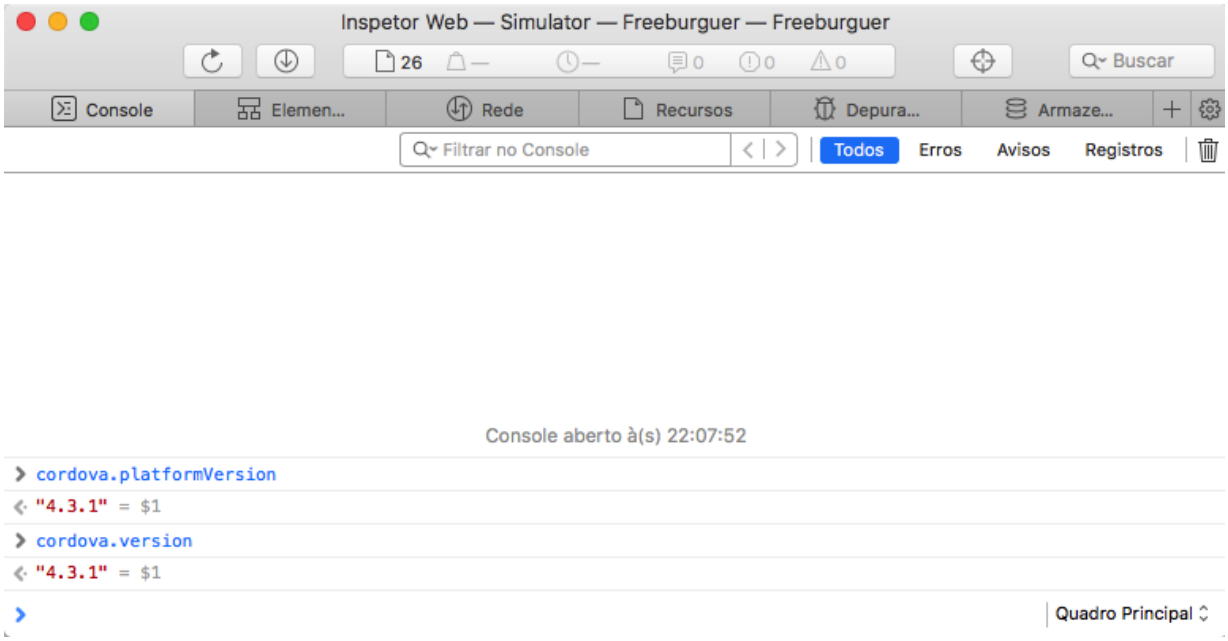


Figura 4.8: Console exclusivo para o aplicativo

Além do console, podemos também mudar o CSS e remover elementos, durante a execução. Em alguns momentos do desenvolvimento, isso pode ser bem útil.

4.8 Uma busca por um pedido específico

Em algum momento, os clientes podem querer voltar ao aplicativo e buscar por um pedido específico. Então, a primeira coisa que vamos fazer é criar uma página elegante para que ele possa inserir o *código* do pedido.

Portanto, criaremos a View `pedido-busca.html`. Ela será bem parecida com a `home`, porém aqui não vamos precisar de nenhum plugin nativo; somente precisamos buscar o pedido no Firebase.

Injetamos no controller `pedidoBusca.js` algumas dependências do AngularJS que vamos usar no código:

```
PedidoBuscaController.$injector = ['$scope', '$rootScope', '$location'];
```

A `$scope` é a variável no nível do controller; `$rootScope` é a variável global; e `$location` é para manipulação da URL e redirecionamentos.

A busca por um pedido segue a mesma ideia da busca por uma empresa, que já fizemos: cada pedido tem um **código** criado com a letra `P` mais cinco dígitos numéricos aleatórios (por exemplo, *P30033*). Ao tocar no `button` de buscar, ele chama uma função `buscar()` que chamará a função `encontrar()` que vai passar o código.

Lembra da seção *A função para encontrar empresas*, na qual comentamos que o Firebase vai mandar um `warn` no navegador reclamando e pedindo uma indexação do atributo **código**? Para isso, dentro das *regras* do banco de dados no painel do Firebase, vamos adicionar:

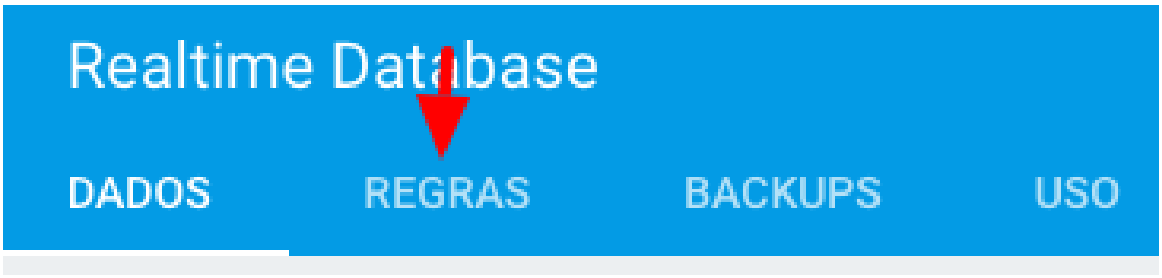


Figura 4.9: Menu no Firebase, dentro de Database

```
"pedidos": { ".indexOn": ["codigo"] },
```

Portanto, o usuário vai inserir o código que ele anotou do pedido, e tocar o `button` que vai chamar a função `buscar()`:

```
this.buscar = function(){
    $rootScope.carregar = true;
    encontrar($scope.codigo_pedido);
}
```

A função `encontrar(codigo)`, por sua vez, inicializará o Firebase ao apontar para a referência de `pedidos`, buscando um pedido na base

que seja igual ao código. Se ele for encontrado, vamos redirecionar o usuário para a rota de informações do pedido; caso contrário, ele tirará o carregamento, vibrará o aparelho e emitirá um alerta ao usuário.

```
function encontrar(codigo){

    var db = firebase.database();
    var pedidos = db.ref('pedidos');

    var query = pedidos
                .orderByChild('codigo')
                .equalTo(codigo)
                .limitToFirst(1);

    query.on('value', function(snapshot){

        if (snapshot.val()){

            var pedido = snapshot.val();
            var keys = Object.keys(pedido);

            $location.path('pedido-info/' + keys[0]);

            $rootScope.carregar = false;

            $scope.$apply();

        }else{

            $rootScope.carregar = false;

            navigator.vibrate(100);

            navigator.notification.alert(
                'Pedido não encontrado.',
                function(){},
                'Ops :(',
                'Ok'
            );
```

```
        }
    });
}
```

4.9 Rastrear um pedido via GPS e um mapa nativo

O dono do Freeburguer falou que cada motoboy tem um GPS instalado enviando os sinais de **longitude** e **latitude** para a base de dados, de tempos em tempos. A ideia é possibilitar que o cliente acompanhe no mapa o local onde o pedido se encontra.

Sendo assim, na tela de acompanhamento do pedido, temos um botão chamado `Rastrear pedido`. Este deve abrir o aplicativo principal de mapa para cada plataforma – *Google Maps* no Android, e o polêmico *Mapas* no iOS.

A primeira coisa que vamos fazer é adicionar o plugin *Inappbrowser*, responsável por abrir o mapa no sistema iOS. Se não usarmos o plugin, a abertura do mapa ainda é possível, mas será muito lenta e insegura.

```
cordova plugin add cordova-plugin-inappbrowser --save
```

Certifique-se de que, dentro do arquivo `config.xml`, existe a linha que libera o uso de URLs com o protocolo `geo:`. Ela é criada por padrão, mas, se não estiver presente, a abertura do mapa pode não funcionar por questões de segurança do Cordova.

A linha que libera o `geo:` é a:

```
<allow-intent href="geo:*" />
```

Para as informações do pedido, dentro do controlador `pedidoInfo.js`, vamos criar a função `navegar(lat, lng)`. Ela recebe a latitude e a longitude, e abre o mapa. Veja no código a seguir:

```
this.navegar = function(lat, lng){

    var plataforma = device.platform;
    plataforma = plataforma.toLowerCase();

    if (lat && lng){
        if (plataforma == 'ios'){
            cordova.InAppBrowser.open('http://maps.apple.com/?
ll='+lat+','+lng+'&q='+lat+','+lng, '_system');
        }else{
            window.location = 'geo:' + lat +"," + lng + '?q='+ lat +"," +
lng;
        }
    }
}
```

Nessa função de `navegar(lat, lng)`, fizemos o seguinte: identificamos primeiro em qual plataforma estamos, iOS ou Android. Para o iOS funcionar corretamente, vamos usar o plugin InAppBrowser, que instalamos logo no começo.

Outro detalhe que precisamos diferenciar é a URL de abertura. No iOS, criamos uma URL usando `http://maps.apple.com` e, no Android, precisamos apenas de `geo:`.

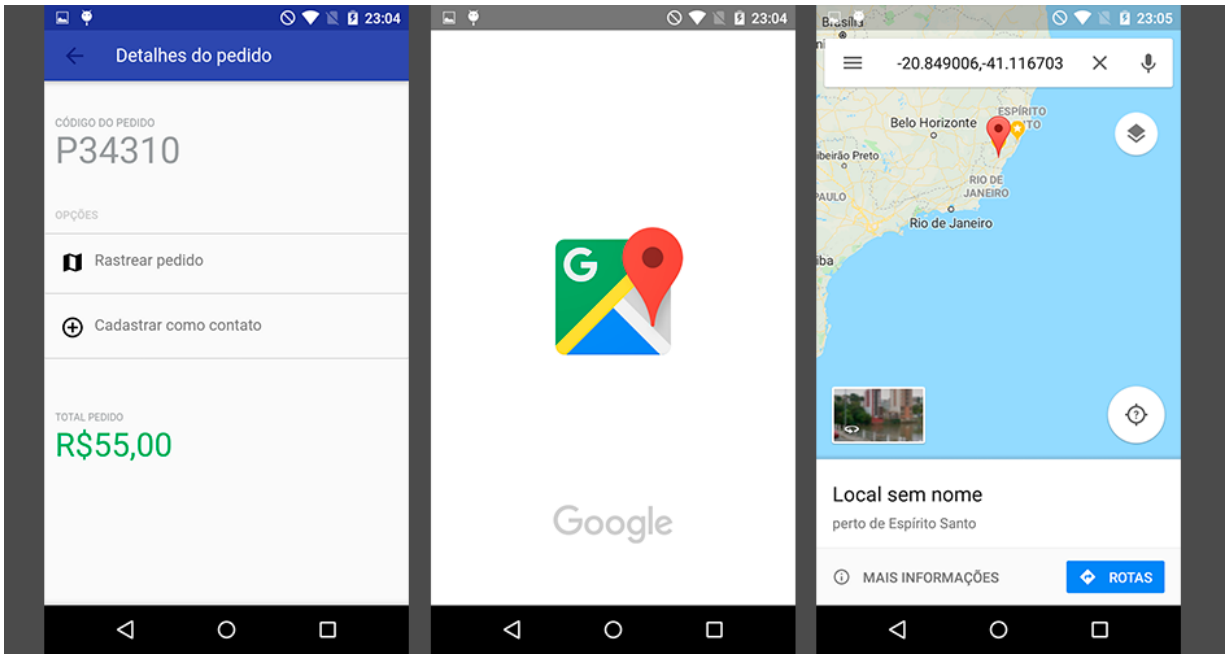
Nos dois casos, para colocarmos um marcador apontando o local exato do mapa, passamos o parâmetro `?q=` com a latitude e a longitude.

- Outros parâmetros de Map Links no iOS – http://apple.co/2xRrm0Z
- Outros parâmetros de Map Links no Android – http://bit.ly/2zaJLtq

Para fechar a funcionalidade, basta agora adicionar a função ao toque do botão:

```
<button type="button" ng-click="PedidoInfo.navegar()">Rastrear
pedido</button>
```

Faça um teste em seu Android ou iOS, assim como vimos no *A primeira versão do aplicativo Freeburguer*.



4.10 Cadastro de uma empresa pelo plugin nativo de contatos

Outra opção que o cliente possui é a de adicionar a empresa na sua lista de contatos. Para isso, vamos usar o plugin `cordova-plugin-contacts`.

12:30
Detalhes do pedido
CÓDIGO DO PEDIDO
P01456
OPÇÕES
Rastrear pedido
Cadastrar como contato
TOTAL PEDIDO
R$ 17,35

Figura 4.11: Imagem ilustrativa tela de checkout

Acesse o diretório raiz do projeto e adicione-o:

```
cordova plugin add cordova-plugin-contacts --save
```

> **iOS 10**
>
> No iOS 10, é obrigatória a passagem de um parâmetro ao adicionar o plugin de contatos, chamado `CONTACTS_USAGE_DESCRIPTION`. Essa variável descreve o motivo que o aplicativo tem para acessar tal recurso na hora de pedir permissão ao usuário.
>
> ```
> cordova plugin add cordova-plugin-contacts --variable
> CONTACTS_USAGE_DESCRIPTION="Seu motivo"
> ```

Para o contato, existe uma série de informações que podemos passar, mas vamos nos atentar apenas ao nome e ao número da empresa. Começaremos criando um novo contato:

```
var contato = navigator.contacts.create({"displayName": "Empresa"});
```

Além do nome, cada contato tem um `Array` de números. É possível indicar qual número será o preferencial do contato no último parâmetro, com `true|false`:

```
var numeros = [];

numeros[0] = new ContactField('work', '99999-1010', false);
numeros[1] = new ContactField('mobile', '99999-2222', true);

contato.phoneNumbers = numeros;
```

Feito isso, vamos de fato salvá-lo no aparelho:

```
contato.save();
```

Pare e realize alguns testes. Caso não se lembre, veja o capítulo *A primeira versão do aplicativo*, onde explicamos como testar com simuladores e em seu dispositivo real. Também já vimos como usar o Google Chrome ou o Safari para inspecionar o projeto. Faça alguns testes e veja se está tudo funcionando!

4.11 Alerta aos usuários offline com o plugin de conexão

Não dá para fazer pedidos offline, por isso, vamos exibir um alerta toda vez que o aplicativo identificar que está sem conexão. Para isso, usaremos o plugin de **Network Information**.

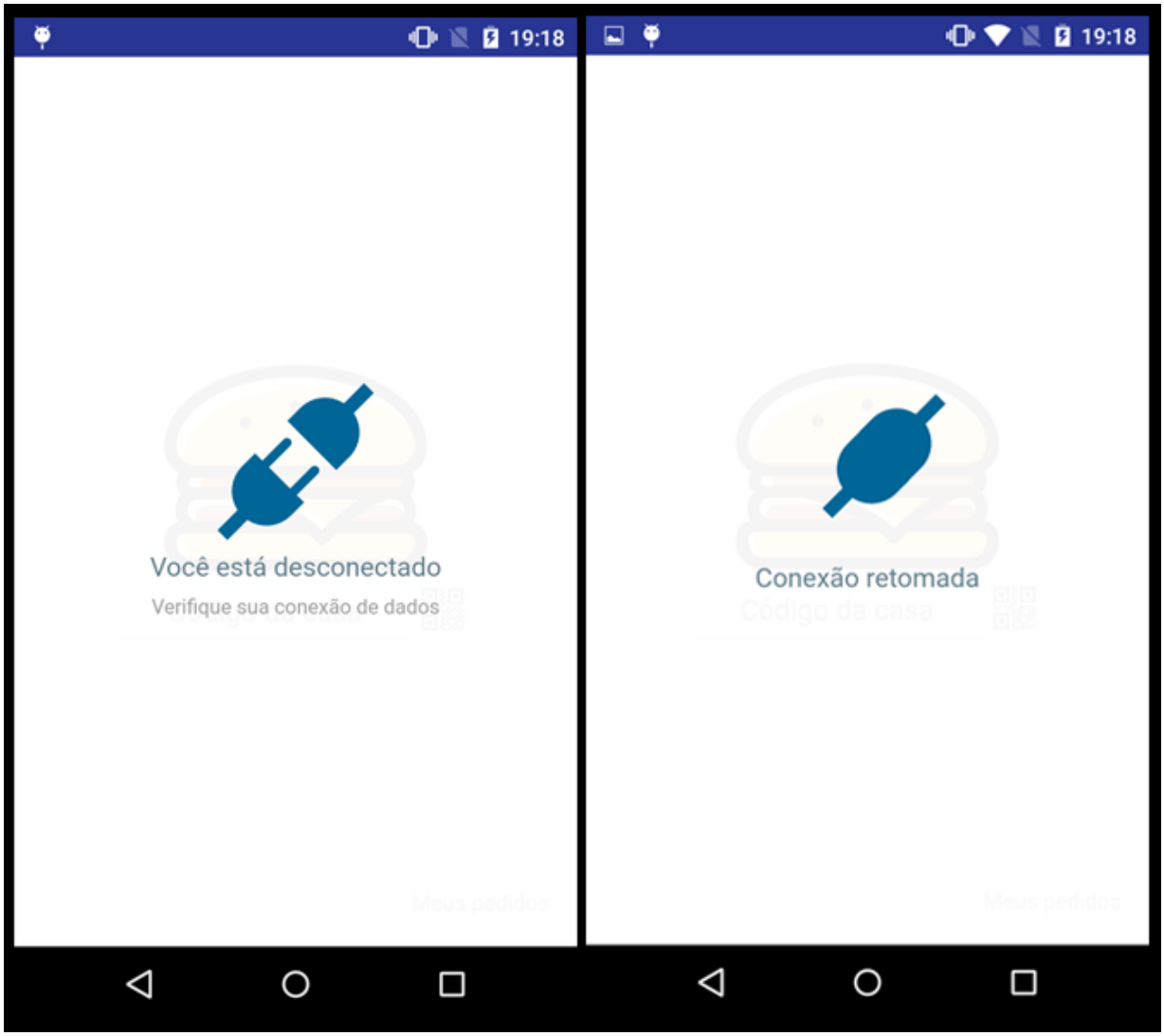


Figura 4.12: Alerta de conexão online e offline

Precisamos adicionar o plugin `cordova-plugin-network-information`:

```
cordova plugin add cordova-plugin-network-information --save
```

No arquivo `bootstrap.js`, vamos adicionar **duas escutas**. A primeira é se estamos *online*: se estivermos, atribuímos `true` para a variável `online`, que vai nos ajudar a exibir (ou não) a tela de mensagem:

```
document.addEventListener("online", onOnline, false);

function onOnline(){
```

```
    $rootScope.online = true;
};
```

Vamos instalar uma escuta para sempre que estivermos *offline*:

```
document.addEventListener("offline", onOffline, false);

function onOffline(){
    $rootScope.online = false;
    $rootScope.alertaOnline = true;
};
```

Com o AngularJS, podemos instalar um observador de variável, chamado `$watch`. Ele observa toda mudança de valor na variável `online`, que recebe `true|false`. Assim, definimos um efeito com CSS usando uma condicional de classes na `div`, com a diretiva `ng-class`:

```
$rootScope.$watch('online', function(){
    if ($rootScope.online){
        $rootScope.alertaOnline = true;
        $timeout(function(){
            $rootScope.alertaOnline = false;
        }, 3000)
    }
});
```

No arquivo `index.html`, antes da `div` com `ng-View`, adicionamos a marcação HTML que exibirá o alerta. Veja a seguir uma versão encurtada do SVG, apenas para um melhor entendimento:

```
<div class="offline-alert" ng-show="alertaOnline" ng-cloak>
    <svg>
        <path class="tomada" ng-class="{'on' : online, 'off' : !online}"
d="" style="fill:#006697"/>
        <path d="" style="fill:#006697"/>
    </svg>

    <p>{{ (!online) ? 'Você está desconectado' : 'Conexão retomada' }}</p>
    <p ng-show="!online">Verifique sua conexão de dados</p>
</div>
```

Para dar um efeito atrativo, usamos um SVG com uma animação que simula uma *tomada de energia*. Ela será animada quando perdermos e restabelecermos a conexão.

Repare que usamos a variável `online` (com valor de `true|false`) para customizar a mensagem para o usuário e adicionar a classe CSS para efetuar a animação, com a condição `ng-class="{'on' : online, 'off' : !online}`.

4.12 O localStorage e o sessionStorage para confirmar a saída

Os donos do Freeburguer são chatos e querem que, apesar de simples, o aplicativo dê resultados. Por isso, pensaram que, se uma pessoa desistir e sair do aplicativo no meio de um pedido, seria interessante quando ela retornar, perguntar se ela deseja continuar aquele pedido. É o que vamos fazer!

Como temos o botão sair em várias Views, vamos criar uma função acessível a todos os controllers. Faremos isso com o `$rootScope`, assim não precisamos criar uma função `sair()` em cada um. No arquivo `bootstrap.js`, criaremos a seguinte função:

```
$rootScope.sair = function(){}
```

Ao tocar em sair na tela, o usuário vai receber uma caixa de diálogo nativa. Para isso, vamos usar o plugin `cordova-plugin-dialogs`. Com o terminal aberto, adicione o plugin ao projeto:

```
cordova plugin add cordova-plugin-dialogs --save
```

No arquivo `bootstrap.js`, faremos:

```
$rootScope.sair = function(){

    navigator.notification.confirm(
```

```
        'Tem certeza que deseja sair?',
         onConfirm,
        'Atenção',
        ['Sim','Não']
    );

    function onConfirm(buttonIndex) {
        if (buttonIndex == 1){
         // Sair...
        }
    }
}
```

Repare que a função de *callback* `onConfirm()` tem como parâmetro o `buttonIndex`, que retornará `1` (resposta para *Sim*), ou `2` (para *Não*).

Se ela for positiva, precisamos saber em qual rota estamos para determinar se devemos gravar no `localStorage` – caso o usuário esteja na tela do cardápio já com alguns itens selecionados. Assim, vamos utilizar uma expressão regular simples:

```
var exp = /(cardapio)/ig;
var location = window.location.hash;
```

Agora, se estivermos na rota de `/cardapio`, gravaremos os dados no `localStorage`:

```
if (location.search(exp) != -1){
    var back = {};

    back.casa_id = sessionStorage.getItem('freeburguer-id');
    back.casa_itens = sessionStorage.getItem('freeburguer-itens');

    localStorage.setItem('freeburguer-back', JSON.stringify(back));
}
```

Repare que usamos também o `sessionStorage` de apoio, porque, quando o usuário entra em uma empresa, guardamos o ID do Firebase em uma sessão chamada `freeburguer-id`; e quando o usuário está selecionando os itens no cardápio, a cada seleção

guardamos os itens em `freeburguer-itens`. Essas duas sessões nos ajudam a redirecionar o usuário caso ele volte ao aplicativo mais tarde.

Por fim, saímos do aplicativo:

```
navigator.app.exitApp();
```

E o **código completo** da função ficou:

```
$rootScope.sair = function(){

    navigator.notification.confirm(
            'Tem certeza que deseja sair?', // Mensagem
            callbackDismiss, // Função de callback
            'Atenção',       // Título
            ['Sim','Não']    // Botões
    );

    function callbackDismiss(buttonIndex){

        if (buttonIndex == 1){

            var exp = /(cardapio)/ig;
            var location = window.location.hash;

            if (location.search(exp) != -1){

                // Variável de apoio
                var back = {};

                // Gravo a ID do firebase da casa de hamburgueres
                back.casa_id = sessionStorage.getItem('freeburguer-id');

                // Gravo os itens até então selecionados
                back.casa_itens = sessionStorage.getItem('freeburguer-
itens');

                // Grava no localStorage
                localStorage.setItem('freeburguer-back',
JSON.stringify(back));
```

```
            }

            navigator.app.exitApp();
        }
    }
}
```

4.13 Evento backbutton do Android

Os aparelhos Android possuem o botão de voltar físico ou virtualizado. Se não o tratarmos quando esse evento ocorre, podemos ter problemas. Isso porque, ao tocarmos em voltar, ele retorna às rotas percorridas seguindo o *History API* do HTML5.

Lembra que, por motivo de organização do código, usamos o arquivo `bootstrap.js` (localizado em `freeburguer -> www -> app -> bootstrap.js`) para fazer inicializações e configurações? Agora, vamos tratar o evento `backbutton` lá também.

```
document.addEventListener("backbutton", function(){

    if ($location.path() == '/'){

            navigator.notification.confirm(
                'Deseja sair do aplicativo?',
                onConfirm,
                'Atenção',
                ['Sim','Não']
            );

            function onConfirm(buttonIndex) {
                if (buttonIndex == 1){
                    navigator.app.exitApp();
                }
            }
    }
    else{
```

```
            $window.history.go(-1);
    }

}, false);
```

Dentro do arquivo `bootstrap.js`, lembre-se de injetar os objetos: `$window`, `$location`, `$rootScope`, `$timeout` na função `run()`. Todos esses recursos são do framework AngularJS, e lidam bem com o ciclo de atualizações que o framework SPA implementa junto ao DOM (*Document Object Model*).

Por exemplo, se usássemos `setTimeout` no lugar do `service`, em algum momento teríamos de forçar o ciclo de atualização da View do AngularJS usando um `$scope.$apply()`, ou `$scope.$digest()`. Recordando que o `run()` ocorre logo após o AngularJS ser carregado por completo, o nosso código ficou da seguinte maneira:

```
.run(function($window, $location, $rootScope, $timeout){
}
```

O `$location.path()` é do AngularJS e retorna a URL atual que estamos no app. Se cairmos na rota `/`, que é a nossa página inicial, enviaremos uma caixa de diálogo de confirmação; caso o código caia no `else`, usaremos o recurso de `history` do navegador com o `$window` – ou seja, voltaremos à página anterior.

Rastreando outros eventos

Com o Cordova, é possível dar um tratamento adequado para os seguintes eventos nativos do sistema operacional:

- `deviceready`: disparado quando o Cordova é totalmente carregado.
- `pause`: disparado quando o aplicativo entra em background, tipicamente quando o usuário troca de aplicativo sem fechar o nosso.

- `resume` : disparado quando o aplicativo deixa o modo background e voltamos ao seu uso.
- `backbutton` : disparado quando tocamos no botão voltar do aparelho.
- `menubutton` : disparado quando tocamos no botão menu do aparelho.
- `searchbutton` : disparado quando tocamos no botão de pesquisa do aparelho.
- `startcallbutton` : disparado quando tocamos no botão de iniciar chamada do aparelho.
- `endcallbutton` : disparado quando tocamos no botão de finalizar chamada do aparelho.
- `volumedownbutton` : disparado quando tocamos no botão de baixar o volume do aparelho.
- `volumeupbutton` : disparado quando tocamos no botão de aumentar o volume do aparelho.

O suporte a esses eventos nas plataformas Android e iOS estão na tabela a seguir:

Evento	Suporte Android	Suporte iOS
`deviceready`	Sim	Sim
`pause`	Sim	Sim
`resume`	Sim	Sim
`backbutton`	Sim	Não
`menubutton`	Sim	Não
`searchbutton`	Sim	Não

Evento	Suporte Android	Suporte iOS
`startcallbutton`	Não	Não
`endcallbutton`	Não	Não
`volumedownbutton`	Sim	Não
`volumeupbutton`	Sim	Não

Os mais utilizados são: o `deviceready`, que diz se o framework foi carregado com sucesso; e o `backbutton`, que trata o botão físico ou virtual de voltar no Android. Já os demais podem ser úteis em alguma funcionalidade específica que você queira implementar.

A sintaxe de escuta desses eventos é o uso de um `addEventListener` ao objeto `document`. Veja um exemplo a seguir:

```
document.addEventListener("pause", tratarPausaApp, false);
```

4.14 Confirmação do pedido

Depois de selecionar os hambúrgueres, o aplicativo vai pedir uma confirmação. Portanto, ao tocar no botão `Fechar pedido`, chamamos a função `fecharPedido()` que criará o pedido no Firebase. Esta será definida no controller do cardápio, no arquivo `cardapio.js`.

```
this.fecharPedido = function(){

    navigator.notification.confirm(
        'Ao avançar você irá confirmar o seu pedido, iremos preparar o seu
lanche e enviar ao motoboy, tem certeza disso?',  // Mensagem
        callbackFecharPedido, // Função de callback
        'Atenção',            // Título
        ['Sim','Não']         // buttonName
   );
}
```

Ainda dentro de `this.fecharPedido`, quando a função de *callback* for acionada, concluímos a lógica para fechar o pedido e adicionar os dados na base de dados:

```
function callbackFecharPedido(buttonIndex){

    if (buttonIndex == 1){

        var pedido = {};
        pedido.codigo  = gerarID();
        pedido.empresa = $routeParams.id_empresa;
        pedido.itens   = $scope.itens;

        firebaseTool.create('/pedidos/', pedido)
        .then(function(pedido_id){
            $location.path('pedido-info/' + pedido_id);
        },
        function(){
            console.error("Erro ao criar pedido");
        });
    }
}
```

A função `gerarID()` é um recurso apenas para gerar um ID aleatoriamente, iniciado com a string `P` mais o valor gerado:

```
function gerarID(){
    return 'P' + Math.random().toString(5).substr(2, 5);
}
```

Ao confirmar um pedido, usamos um recurso chamado `firebaseTool.create()`. Ele funciona como um prestador de serviços que facilita a inserção de dados na base do Firebase, e encontra-se no arquivo `freeburguer -> www -> app -> services -> firebaseTool.js`.

O recurso que nomeei de `firebaseTool` é chamado no AngularJS de *Service*. Sua estrutura tem uma escrita bem parecida com a de um controlador:

```
(function() {
    angular.module('app').factory('firebaseTool', firebaseTool);

    function firebaseTool(){
    }
})();
```

Da mesma forma que injetamos outras funcionalidades do AngularJS, quando formos usar o `firebaseTool`, teremos também de injetar dentro do controlador pelo nome definido (no nosso caso, `firebaseTool`).

Por dentro do arquivo firebaseTool.js

Dentro do serviço que criamos, encapsulamos duas funções dentro da variável que chamamos de `service` (pode ser qualquer nome):

```
var service = {};

service.create = function(url, objeto){}
service.set = function(url, objeto){}

return service;
```

Ao final, retornamos a variável com `return`. Assim, quando formos utilizar o `firebaseTool`, podemos chamar tanto o `create()` quanto o `set()`.

A função `create(url, objeto)` recebe `url` como o caminho em que vamos armazenar. O outro parâmetro é o `objeto` que contém os dados. Dentro dessa função, vamos usar um componente chamado `$q` para criar uma *promise* (promessa). **Lembre-se** de injetar o componente na função `firebaseTool()`.

```
var deferred = $q.defer();
```

Continuando a função, vamos criar uma nova chave do Firebase, gerada automaticamente pela plataforma – como um ID autoincremento:

```
    var nova_chave = firebase.database().ref().child(url).push().key;
```

Com essa chave armazenada na variável `nova_chave`, criamos uma variável de apoio chamada `data` para armazenar o `objeto` com as informações que vamos gravar:

```
var data = {};
data[url + nova_chave] = objeto;
```

Agora, mandamos o Firebase inserir com o comando `update`, e recolhemos a *promessa* JavaScript que ele retorna:

```
var promise = firebase.database().ref().update(new_data);
```

Com essa *promessa*, podemos tratar a função `.then()` para saber se tudo ocorreu certo ou não:

```
promise.then(function(){
    deferred.resolve(new_key);
},
function(error){
    deferred.reject("Problemas ao inserir no DB");
});
```

Repare que, caso tudo tenha ocorrido bem e os dados tenham sido inseridos, ainda queremos resolver a nossa outra promessa criada logo no início da função, para isso usamos os métodos `resolve` e `reject`, portanto, caso os dados tenham sido inseridos, nós fazemos um `resolve` retornando a nova chave criada, em seguida, se ocorreu um erro, usamos o `reject` e também retornamos uma mensagem de erro.

O QUE É UMA PROMISE?

Promise é um objeto usado para um processamento assíncrono. Uma promise (de "promessa") representa um valor que pode estar disponível agora, no futuro ou nunca.

Ao final da função `create`, retornamos a promessa que chamamos de `deferred`:

```
return deferred.promise;
```

A outra função `set()` é mais simples. Ela também recebe o caminho `url` e a variável com as informações para armazenar, chamada `objeto`. Porém, agora não precisamos de uma chave incrementada do Firebase, portanto, só usamos a seguinte linha:

```
firebase.database().ref(url).set(objeto);
```

Essa linha anterior será suficiente para atualizar ou criar os dados na base. O código todo do `service` ficou:

```
(function() {
    angular.module('app').factory('firebaseTool', firebaseTool);

    function firebaseTool($q){

        var service = {};

        service.create = function(url, objeto){

            var deferred = $q.defer();

            var new_key = firebase.database().ref().child(url).push().key;

            var new_data = {};
            new_data[url + new_key] = objeto;

            var promise = firebase.database().ref().update(new_data);

            promise.then(function(){
                deferred.resolve(new_key);
            },
            function(error){
                deferred.reject("Problemas ao inserir no DB");
            });

            return deferred.promise;
```

```
        }

        service.set = function(url, objeto){
            return firebase.database().ref(url).set(objeto);
        }

        return service;
    }
})();
```

Não se esqueça de carregar o arquivo no `index.html`. Você também deve injetar a dependência `firebaseTool` toda vez que for usá-la em um controlador.

> Você pode baixar o aplicativo pronto até aqui em http://bit.ly/cordova-livro. O projeto é o **freeburguer-cap5**.

4.15 Revisão

Neste capítulo, trabalhamos com plugins nativos de forma efetiva. O Cordova e o PhoneGap possuem uma gama muito grande de plugins disponíveis e as opções são bastante atraentes. No momento em que escrevo este livro, são cerca de 3.084 plugins (dados extraídos do site oficial do Cordova em janeiro de 2018) específicos para Android e iOS. Basta procurar por `cordova-plugin-<NOME>` ou `phonegap-plugin-<NOME>` no repositório oficial (https://www.npmjs.com) ou, mais fácil que isso, acessar http://cordova.apache.org/plugins.

Já falamos de configurações e design, e construímos as principais funcionalidades do app. Nos dois próximos capítulos, falaremos de detalhes importantes da plataforma Android e iOS, de como gerar o seu aplicativo assinado, e opções exclusivas de cada plataforma em

relação à WebView. Também veremos como funciona cada loja de aplicativos por dentro e boas práticas de publicação.

Capítulo 5
Detalhes exclusivos da plataforma Android

Agora que já temos o aplicativo pronto, daremos mais atenção neste capítulo à plataforma Android. Nossa primeira tarefa é nos certificar de ter a plataforma adicionada ao projeto. Dentro da raiz do projeto Freeburguer, execute o comando a seguir:

```
cordova platform add android
```

5.1 Emulação

Para emular o Android, as duas melhores opções são usar: o Android Virtual Devices (AVDs) ou o Genymotion.

Usando Android Virtual Devices (AVDs)

Android Virtual Devices (AVDs) é uma definição que permite definir características de um celular ou de um tablet Android que você deseja simular no Android Emulator. Um AVD contém um perfil de hardware, uma imagem de sistema, uma área de armazenamento, uma aparência e outras propriedades.

No capítulo *Configurando seu ambiente de desenvolvimento*, certificamos de deixar tudo configurado para que agora possamos utilizar o Android SDK. Vamos abrir o gerenciador do Android com o comando `android` no terminal. Na barra superior do computador, clique em `Tools -> Manage AVDs .`

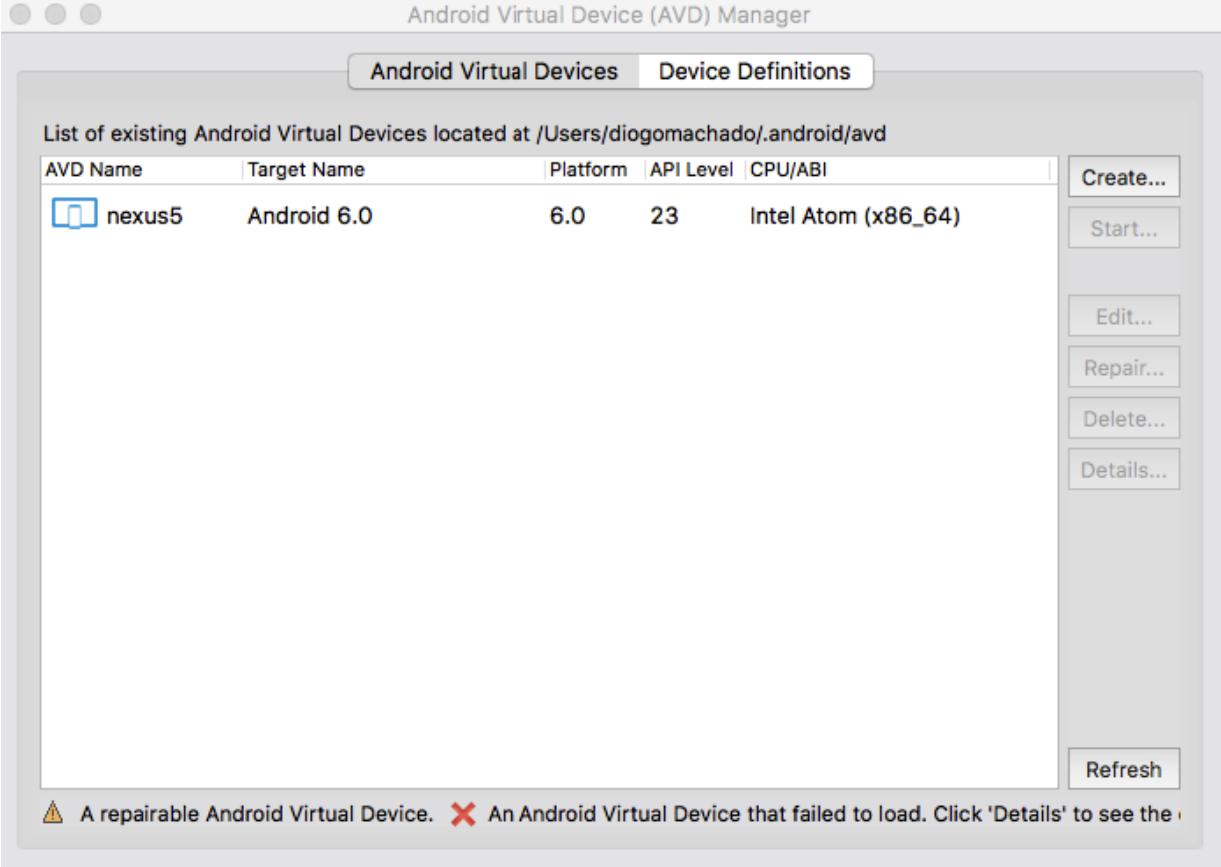


Figura 5.1: AVD Manager

O AVD Manager é pré-carregado com alguns perfis de hardware na aba `Device Definitions`, como o aparelho *Nexus*. Você pode definir e importar outros perfis de hardware conforme sua necessidade.

Em geral, a maioria dos perfis pré-carregados já nos atenderão muito bem. Por isso, vamos criar um aparelho com o perfil *Nexus 5*, mas se você escolher outro perfil ou configurar um manualmente, não terá problemas.

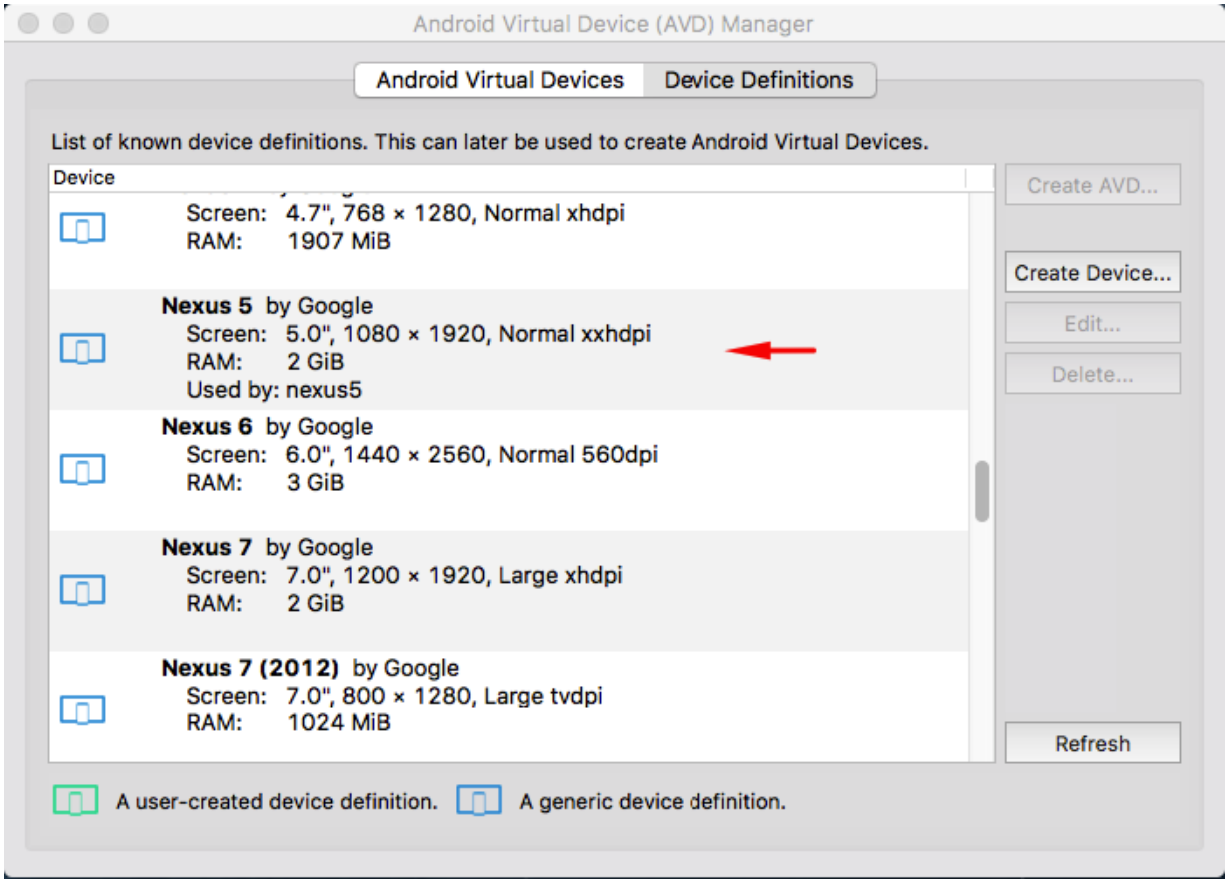

Android Virtual Device (AVD) Manager
Android Virtual Devices
Device Definitions
List of known device definitions. This can later be used to create Android Virtual Devices.
Device
Screen: 4.7", 768 × 1280, Normal xhdpi
RAM: 1907 MiB
Nexus 5 by Google
Screen: 5.0", 1080 × 1920, Normal xxhdpi
RAM: 2 GiB
Used by: nexus5
Nexus 6 by Google
Screen: 6.0", 1440 × 2560, Normal 560dpi
RAM: 3 GiB
Nexus 7 by Google
Screen: 7.0", 1200 × 1920, Large xhdpi
RAM: 2 GiB
Nexus 7 (2012) by Google
Screen: 7.0", 800 × 1280, Large tvdpi
RAM: 1024 MiB
Create AVD...
Create Device...
Edit...
Delete...
Refresh
A user-created device definition.
A generic device definition.

Figura 5.2: AVD Manager perfil Nexus

Clique em `Device Definitions`, escolha a opção `Nexus 5`, e clique em `Create AVD`. A janela que se abrirá é a de configurações do dispositivo. Explicarei as partes mais importantes a seguir.

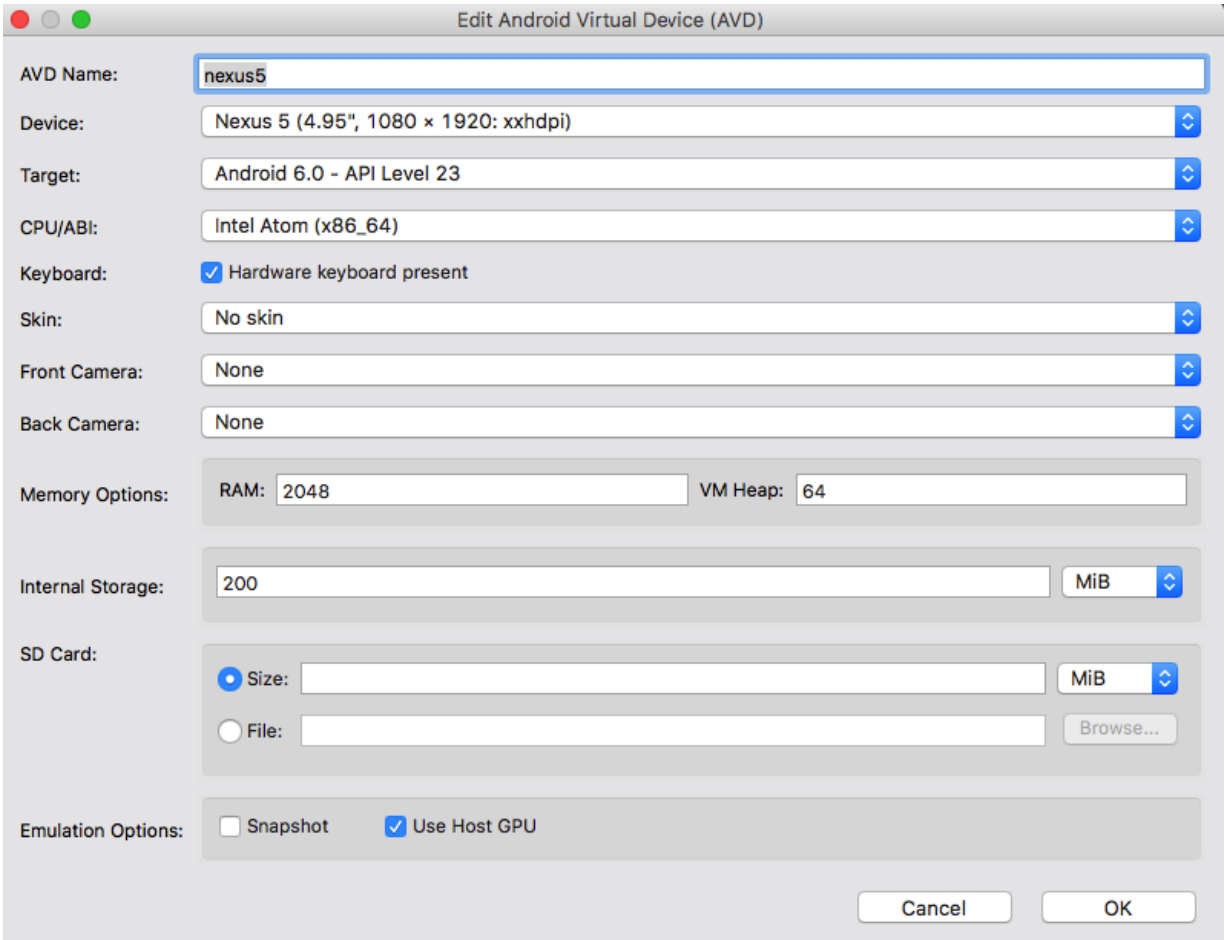


Figura 5.3: Janela de configurações do AVD

- **AVD Name:** use um nome fácil, isso vai facilitar na hora de rodar o comando de emulação. Vamos chamar de `nexus5`.
- **Target:** esse parâmetro diz respeito à versão do Android. A imagem do sistema fica a seu critério aqui, e vai depender de quais versões você tem instaladas. No exemplo, escolhemos o `Android 6.0 - API Level 23`.
- **CPU/ABI:** aqui é a arquitetura do CPU e, segundo o Google, arquiteturas x86 são mais rápidas para emular.
- **Skin:** esse parâmetro tem a ver com a densidade da tela que queremos emular. Lembra que falamos desse assunto na seção *O tamanho de um pixel nas diferentes resoluções de tela*?

Como essa opção trabalha com o tamanho e a densidade emulada, se você estiver usando um computador com tela pequena, (como é o meu caso, um Macbook com tela 13), essas opções podem não ser muito úteis, já que o sistema Android será cortado na tela do emulador devido à alta resolução. Por isso, vamos marcar a opção `No skin` para conseguirmos ver toda a aplicação na tela da nossa máquina.

	Low density (120), *ldpi*	Medium density (160), *mdpi*	High density (240), *hdpi*	Extra-high-density (320), *xhdpi*
Small screen	**QVGA (240x320)**		480x640	
Normal screen	**WQVGA400 (240x400)** **WQVGA432 (240x432)**	**HVGA (320x480)**	**WVGA800 (480x800)** **WVGA854 (480x854)** 600x1024	640x960
Large screen	**WVGA800** (480x800)** **WVGA854** (480x854)**	**WVGA800* (480x800)** **WVGA854* (480x854)** 600x1024		
Extra-Large screen	1024x600	**WXGA (1280x800)**† 1024x768 1280x768	1536x1152 1920x1152 1920x1200	2048x1536 2560x1536 2560x1600

Figura 5.4: Tabela do Android de densidades

Por último, em `Emulation Options`, vamos marcar a opção `Use host GPU`. Esta vai ajudar o emulador a funcionar mais fluido usando o GPU da sua máquina. A diferença que essa opção faz é bastante notável.

Outras configurações podem ser feitas, como tamanho de memória, emulação da câmera frontal etc. Isso vai depender da sua necessidade. Para o nosso teste, os parâmetros anteriores são suficientes, então, clique em `Ok` para terminar de criar o AVD.

Agora que já temos um dispositivo criado, podemos testar nosso aplicativo. Use o comando de listar para ver os dispositivos virtuais e físicos disponíveis para teste:

```
cordova run android --list
```

Quando você tem um aparelho ligado à sua máquina via USB, ele vai listar em *android devices*; os demais, serão *Android Virtual*

Devices. O resultado esperado com o comando de listar é algo como mostro a seguir:

```
Available android devices:
0422213466
Available android virtual devices:
Android_Accelerated_Nougat
Android_ARMv7a_Nougat
nexus5
```

Repare que, em `Available android virtual devices:` , temos o nosso `nexus5` . Basta emular usando seu nome como `target` :

```
cordova run android --target=nexus5
```

Após o comando anterior, o projeto será compilado, o emulador será inicializado, e o aplicativo será instalado no dispositivo virtual e executado. Se tudo der certo, você verá um **BUILD SUCCESSFUL** na linha de comando. Caso algum erro ocorra, fique atento ao terminal de comando e veja se ele é referente a algum plugin ou configuração.

Ao final, temos nosso aplicativo em execução. Algumas das grandes vantagens de usar AVDs são a capacidade de customização do hardware e por eles serem a opção oficial do Android SDK. A maior desvantagem é que eles são lentos; se você tiver pouca memória e um processador lerdo, a emulação e a configuração dos AVDs podem ser um processo bem cansativo.

Genymotion

O Genymotion é um emulador incrivelmente rápido, eficiente em termos de memória, e executa o sistema operacional Android de uma forma muito realística. Na verdade, ele é mais que um emulador Android; ele é uma plataforma completa de virtualização para desenvolvedores Android testarem seus aplicativos.

Pode ser uma ótima alternativa caso os AVDs oficiais do Android Studio fiquem muito lento em seu ambiente de desenvolvimento.

Você pode baixá-lo em https://www.genymotion.com; basta criar uma conta para fazer o download.

O Genymotion *virtualiza* o Android usando o software Virtualbox. Portanto, se você não tiver instalado no computador, baixe e instale-o, em https://www.virtualbox.org.

Após instalado, criar aparelhos virtuais é fácil. Basta clicar em `Add` e selecionar um dos aparelhos. É possível ainda filtrar pela versão do Android.

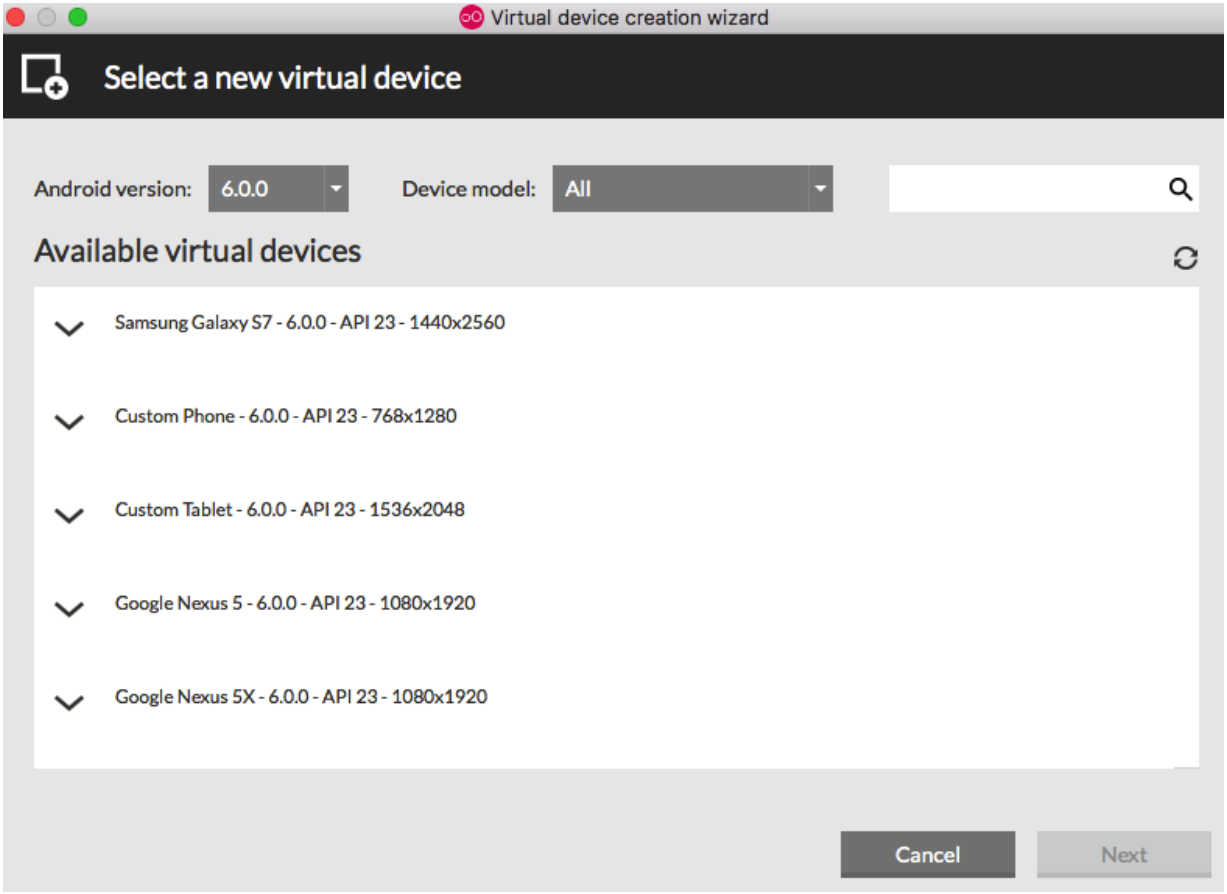

Virtual device creation wizard
Select a new virtual device
Android version:
6.0.0
Device model:
All
Available virtual devices
Samsung Galaxy S7 - 6.0.0 - API 23 - 1440x2560
Custom Phone - 6.0.0 - API 23 - 768x1280
Custom Tablet - 6.0.0 - API 23 - 1536x2048
Google Nexus 5 - 6.0.0 - API 23 - 1080x1920
Google Nexus 5X - 6.0.0 - API 23 - 1080x1920
Cancel
Next

Figura 5.5: Genymotion tela de adição

Após criar, basta dar `Start` no aparelho para executar o simulador. Depois disso, assim como os AVDs, o aparelho estará disponível para testarmos, e poderemos rodar o comando para listar os dispositivos:

```
cordova run android --list
```

Geralmente, o Genymotion vai criar um dispositivo *android device* na listagem com um IP, como `192.168.57.101:5555`. Basta usar esse número como `target`:

```
cordova run android --target=192.168.57.101:5555
```

De fato, a grande vantagem do Genymotion é ser menos complicado e rodar mais rápido que os tradicionais AVDs. Uma desvantagem é que alguns recursos na emulação só estão disponíveis na versão paga, mas, mesmo assim, ele ainda é uma ótima opção.

> **Você sabia**
>
> É possível depurar no Google Chrome um aplicativo em execução no Genymotion. É só seguir os mesmos passos da seção *Como usar o Google Chrome para inspecionar o projeto em execução*.

Na prática, tenho preferência pelo Genymotion, por ser mais rápido de configurar e executar os simuladores em comparação aos AVDs. Prefira AVDs quando você tiver de configurar um hardware mais específico.

5.2 WebViews antigas e o projeto Crosswalk

Estamos desenvolvendo um aplicativo híbrido, que será executado dentro de uma WebView. Acontece que a WebView dos aparelhos com Android 4.x são muito ruins e desatualizadas.

Segundo Sérgio Lopes, no livro *Aplicações mobile híbridas com Cordova e Phonegap*, a WebView passou a ser um componente independente do sistema operacional somente na versão do Android

5.0. Só que, ainda, temos muitos aparelhos usando versões abaixo disso. Dados de dezembro de 2017 mostram que o Android 6.0 (Marshmallow, 23) é o mais usado, com 29,7%; seguido do 5.1 (Lollipop, 22), com 20,2%. O Android 4.4 (KitKat, 19) está com 13,4%, como mostra a imagem a seguir (http://bit.ly/2zuRW45):

Version	Codename	API	Distribution
2.2	Froyo	8	0.1%
2.3.3 - 2.3.7	Gingerbread	10	1.7%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	1.6%
4.1.x	Jelly Bean	16	6.0%
4.2.x		17	8.3%
4.3		18	2.4%
4.4	KitKat	19	29.2%
5.0	Lollipop	21	14.1%
5.1		22	21.4%
6.0	Marshmallow	23	15.2%

Dados coletados durante um período de 7 dias encerrado em 1º de agosto de 2016.
Todas as versões com menos de 0,1% de distribuição não foram exibidas.

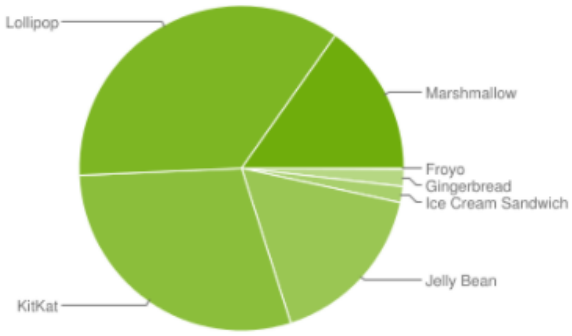


Figura 5.6: Dados de uso do Android no mundo

Para isso, existe o projeto **Crosswalk**. Este empacota uma WebView baseada no Chromium, junto com a nossa aplicação Cordova ou PhoneGap. O ponto negativo é que o Crosswalk vai embutir um Chromium inteiro dentro da sua aplicação, aumentando muito o tamanho do seu `apk`. Por outro lado, o ponto positivo é que o Chromium traz consigo as APIs mais atuais, garantindo um suporte em Androids antigos.

Vamos ver como usar o projeto. O uso do Crosswalk fica a seu critério; só você deve decidir quando o uso é justificável.

Usando o Crosswalk

Para usá-lo, é simples; basta adicionar o plugin ao seu projeto:

```
cordova plugin add cordova-plugin-crosswalk-webview --save
```

Para gerar, rodamos o comando a seguir:

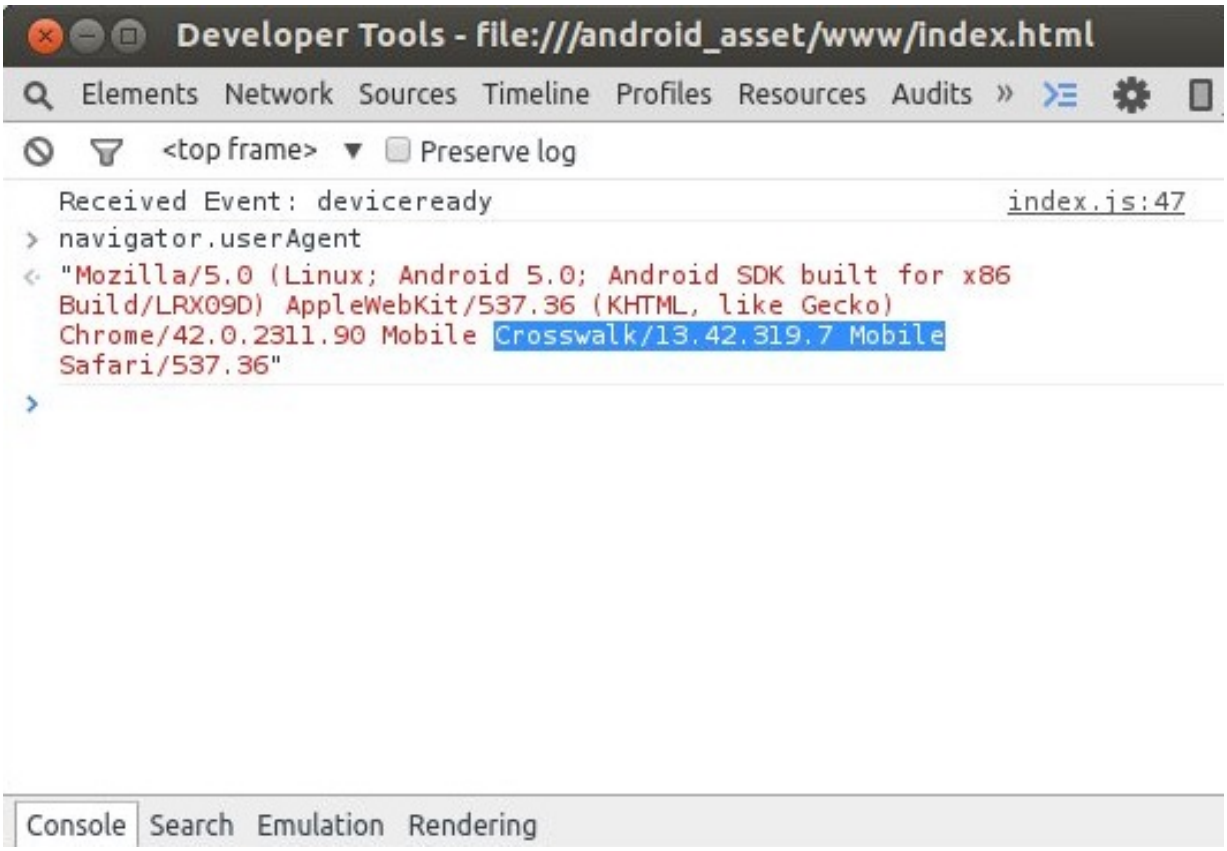
```
cordova build android
```

Repare que teremos dois `apk` **diferentes**. Isso porque o binário do *Chromium* muda de acordo com a arquitetura. Eles serão gerados no diretório de build `freeburguer -> platforms -> android -> build -> outputs -> apk` , com os nomes:

- `android-x86-debug.apk`
- `android-armv7-debug.apk`

> O nome dos pacotes gerados no Android serão sempre `android` mais o restante. Com Crosswalk, junta-se a arquitetura `x86` ou `armv7` e o estado do `apk` ; quando em desenvolvimento, será `-debug` ; e quando em produção usando o comando de `build` (`cordova build android --release`), será `-release` .

Para saber se o WebView foi injetado corretamente, você pode inspecionar o projeto com o Google Chrome e, no console, inserir o comando `navigator.userAgent` . O resultado deverá ser algo parecido com a figura a seguir:



PROJETO CROSSWALK NÃO SERÁ MAIS ATUALIZADO

A equipe do projeto Crosswalk anunciou que ele não será mais atualizado (última versão estável é a 23), porque as coisas evoluíram e não há mais necessidade de manter o projeto. Com o tempo, o Android 4.x será menos utilizado, e as novas versões já são bem amparadas por uma boa WebView e por outras tecnologias. Há também o crescimento do Progressive Web Apps. De qualquer forma, você pode continuar usando a versão mais estável.

5.3 Assinatura

Se você nunca publicou nada no Google Play, você precisa criar uma chave de assinatura. Para criar um par de chaves para assinar o aplicativo, vamos usar uma ferramenta disponível no terminal de comando que vem junto com o Java JDK, chamada `keytool`. O comando a seguir é suficiente:

```
keytool -genkey -v -keystore freeburguer.keystore -alias freeburguer
-keyalg RSA -keysize 2048 -validity 10000
```

Esse comando lhe perguntará uma série de informações, como o seu nome, local, uma senha para proteger (para o livro, escolhemos 123456), a chave e outros. Preencha tudo corretamente.

> **CUIDADO**
>
> Guarde essa chave com muito cuidado. Sem ela, não será possível atualizar o aplicativo nunca mais. Também não a repasse para outras pessoas por segurança.

Ainda sobre o comando `keytool`, veja a seguir uma imagem para explicar melhor.

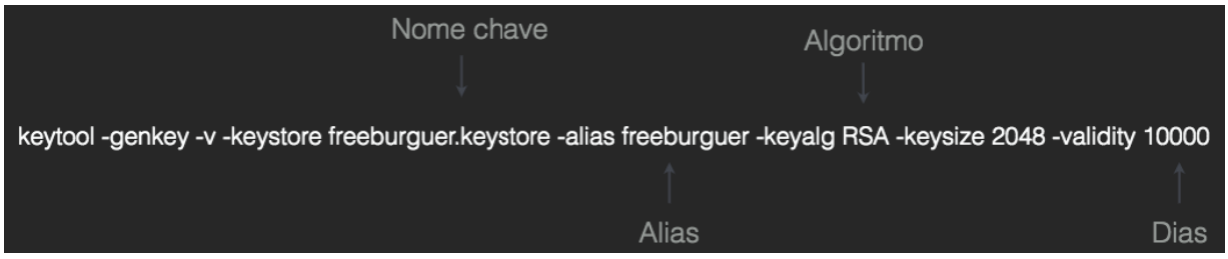


Figura 5.8: Comando key tool

Agora, com a chave gerada, vamos preparar um arquivo chamado `build.json`. Esse arquivo deve ficar na raiz do projeto, no diretório `freeburguer`. Ele será responsável por mapear a chave para assinar o aplicativo quando executarmos o comando de build:

```
{
    "android": {
```

```
      "release": {
          "keystore": "./freeburguer.keystore",
        "storePassword": "123456",
          "alias": "freeburguer",
        "password" : "123456",
        "keystoreType": ""
         }
    }
}
```

Enquanto não gerarmos o aplicativo usando `--release`, o `apk` será gerado no diretório `freeburguer -> platforms -> android -> build -> outputs -> apk` com o nome de `android-debug.apk`. Depois de criarmos e rodarmos o comando para assinar o aplicativo, o `apk` terá o nome de `android-release.apk`. A diferença deles é mostrada na imagem a seguir.

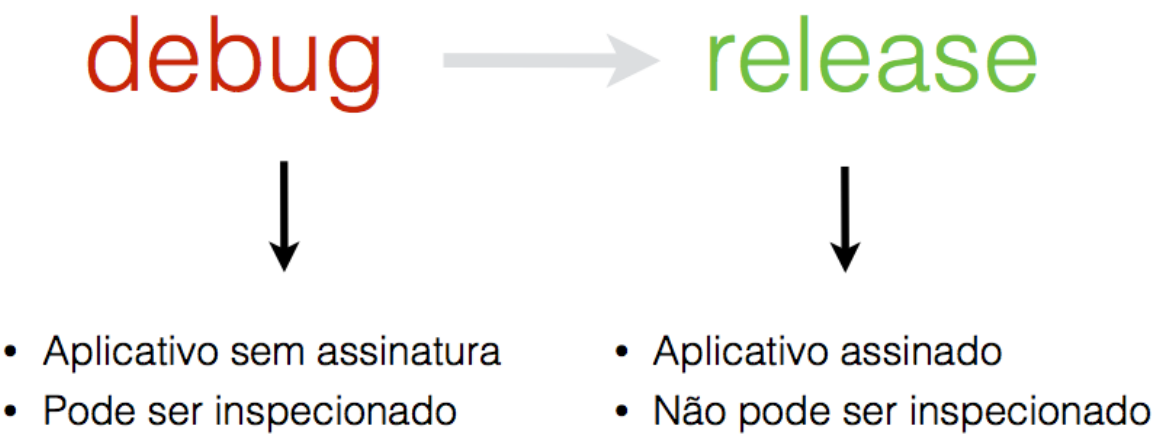


Figura 5.9: Diferenças dos apks

Estamos prontos para publicar! Agora, vamos adicionar os parâmetros `--release` para gerar o `apk` final, e `--buildConfig` para usar as configurações do arquivo `build.json`:

```
cordova build android --release --buildConfig
```

Ao final, você pode localizar o arquivo em `freeburguer -> platforms -> android -> build -> outputs -> apk -> android-release.apk`. Ele será

usado para subir na loja.

> **O caso Crosswalk**
>
> Se você optar por usar o projeto Crosswalk, será necessário subir os dois `apks` gerados. Lembre-se de que o Google Play vai distribuir corretamente esses arquivos de acordo com a arquitetura do usuário (`x86` ou `armeabi-v7a`); fique tranquilo.

5.4 Subindo o apk para o Google Play

Para subir o app, devemos ir ao https://play.google.com/apps/publish. Se você nunca pagou sua conta de desenvolvedor, o custo será de 25 dólares, pago uma única vez.

O Google recentemente simplificou muito a plataforma de publicação. Dentro do painel, clique em `Criar app` , insira o nome `Freeburguer` e finalize com `Criar` . Neste momento, seremos redirecionados para `Detalhes do app` .

Você deve preencher **com muita calma** todos os campos que possuem um asterisco. As informações podem ser preenchidas em outros idiomas e, se você for distribuir para outros países, é bem interessante fornecê-las na língua nativa respectiva.

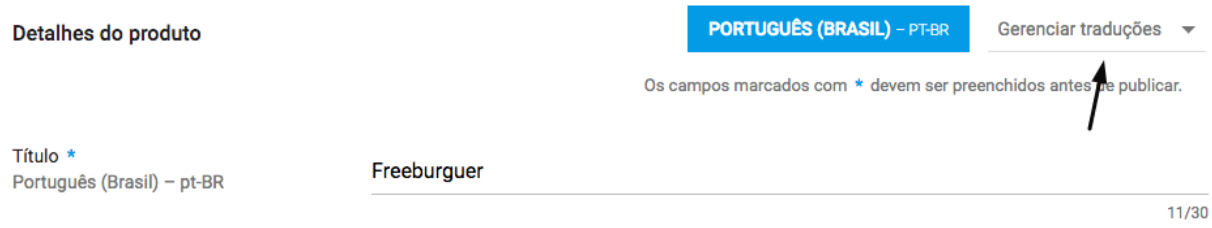


Figura 5.10: Opção de tradução do título e outras informações

Perceba que, para publicarmos um aplicativo, devemos cumprir algumas etapas, e passar pelas abas marcadas com um ícone até que a plataforma esteja satisfeita com as informações obrigatórias, imagens, termos, valor e outras informações pertinentes.

Não existe uma ordem para preencher os dados, mas, como sugestão, você pode preencher desta forma:

1. Detalhes do app;
2. Classificação do conteúdo;
3. Preços e distribuição;
4. Versões de apps (última etapa).

Deixamos a fase de enviar o `apk` por último, porque vamos precisar de todos os outros campos concluídos para isso.

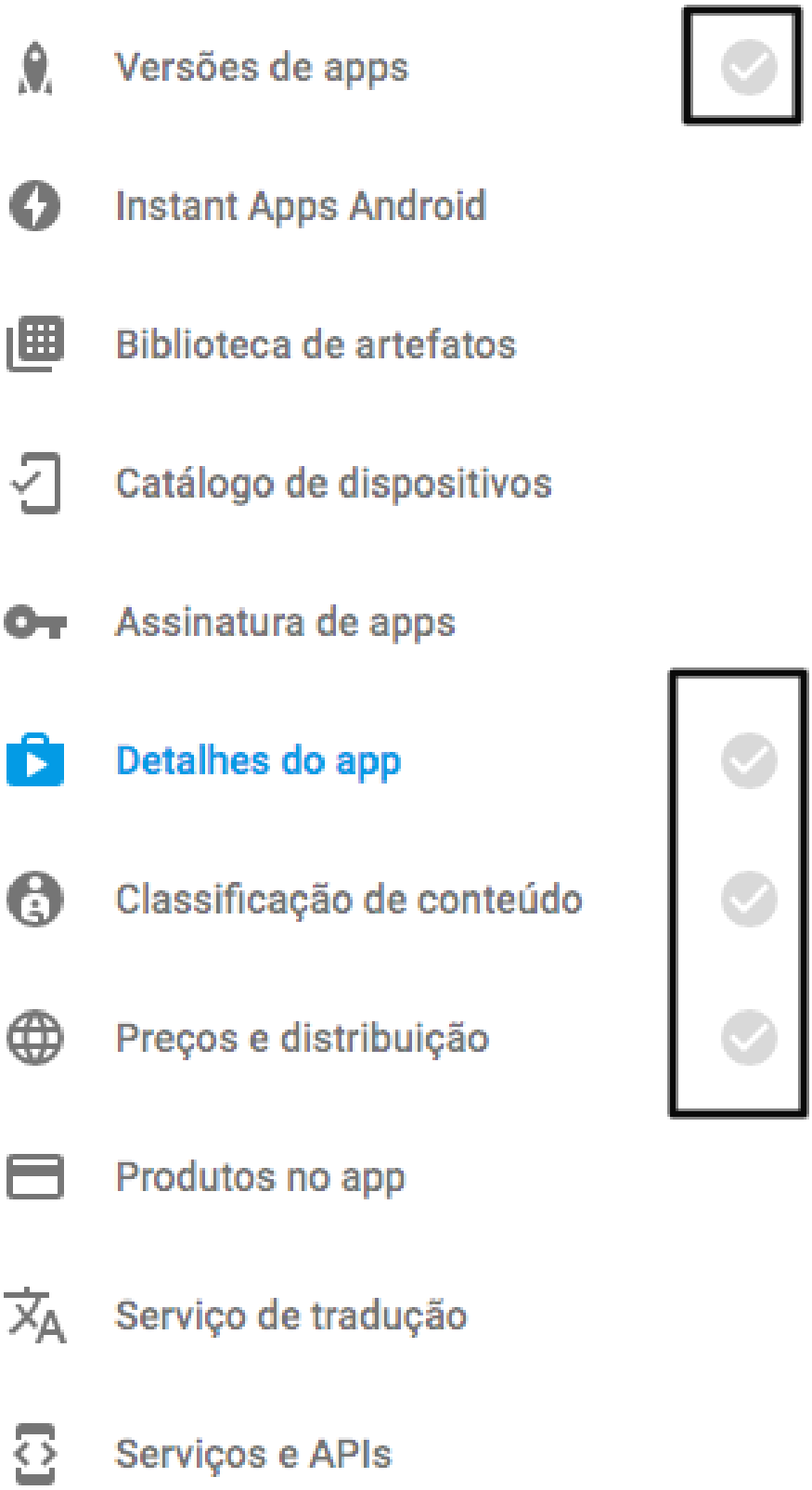


Figura 5.11: Menu de opções dentro do Google Play Console

Na última etapa sugerida, a aba `Versões de apps`, é onde subimos o `apk` gerado. Ao clicar nela, temos uma janela muito bem dividida para as fases *Alfa*, *Beta* e *Produção*. É claro que você pode publicar um aplicativo direto em `Produção` – apesar de não ser recomendado, dependendo da proporção do projeto.

Freeburguer
Rascunho
SALVAR RASCUNHO
Por que não posso publicar?
Gerencie os APKs do seu app, analise o histórico da versão e lance seu app para produção, Alfa ou Beta. Saiba mais
Produção
GERENCIAR PRODUÇÃO
Você tem uma versão em produção que não foi lançada
EDITAR VERSÃO
Beta
GERENCIAR BETA
Adicione APKs à faixa Beta se quiser que o app esteja disponível para teste Beta aberto ou fechado.
Alfa
GERENCIAR ALFA
Adicione APKs à faixa Alfa se quiser que o app esteja disponível para teste Alfa aberto ou fechado.

Figura 5.12: Gerenciador de versões

Agora vamos subir o nosso `apk` de *release* para um nível `Alfa`, ou seja, distribuir para alguns usuários testarem. Dentro da aba `Versões de apps`, clicamos em `Gerenciar Alfa`, e depois no botão azul `Criar uma versão`. Em seguida, selecionamos `Enviar APK` e o arquivo `android-release.apk` gerado anteriormente.

Em seguida, vamos preencher as informações da versão e escolher uma versão no campo `Nome da versão`. Lembre-se de que será sugerido um nome para versão baseado na versão do `apk`, configurada no arquivo `config.xml` do projeto.

Política de privacidade

Algumas permissões do aplicativo podem precisar de uma política de privacidade. Se seu aplicativo não tem um site próprio, uma solução de graça e eficiente é criar um repositório no GitHub e usar o GitHub Pages.

Ao final de todo processo, clique em `Publicar`. O Google informará que, dentro de algumas horas, o aplicativo estará disponível. Geralmente, o Google tem sido bem rápido na publicação; às vezes, em questão de 20 a 30 minutos, o aplicativo já está no ar.

O marketing do seu aplicativo

Dentro da aba `Detalhes do app`, na seção `Recursos gráficos`, existe uma área para upload de `Capturas de tela`, `Ícone de alta resolução`, `Gráfico de recursos` (aquele que aparece no topo do aplicativo quando não adicionamos um vídeo) e outras imagens não obrigatórias.

Esses recursos foram criados para nos ajudar a *vender* a ideia ao usuário, então, a **qualidade** deles pode ser decisiva entre o usuário instalar ou deixar a página do seu aplicativo. Se você, como desenvolvedor, não for capaz de produzir essas imagens – o que é comum –, contrate um profissional para fazer.

A plataforma diz tudo bem detalhado em termos de tamanhos e formatos. Se você estiver fechando um contrato, já deixe claro se você produz esse tipo de material para o seu cliente.

5.5 Update e downgrade da biblioteca Android do Framework

Cada plataforma do Cordova tem sua biblioteca. Essa biblioteca possui uma versão e o devido suporte aos plugins de acesso nativo. Com o tempo, você pode querer atualizar ou voltar a uma versão mais antiga da biblioteca do Cordova ou do PhoneGap, por motivos de atualizações ou porque um plugin parou de funcionar com a versão nova.

Para atualizar, basta acessar a aplicação em seu diretório raiz e rodar o comando:

```
cordova platform update android
```

É possível também ir para uma versão específica, usando o arroba e a versão:

```
cordova platform update android@5.0.0
```

Você pode olhar o blog do Cordova para saber as novidades sobre suporte na biblioteca Android e outros lançamentos: http://cordova.apache.org/blog.

> Você pode baixar o aplicativo pronto até aqui em http://bit.ly/cordova-livro. O projeto é o **freeburguer-cap5**.

5.6 Revisão

Neste capítulo, conhecemos detalhes específicos da plataforma Android, e vimos algumas opções de como emular o nosso aplicativo utilizando AVDs e Genymotion. Também conhecemos o projeto para Crosswalk, que ajuda a dar suporte à WebView de aparelhos Android mais antigos, sendo sua a decisão de usar ou não.

Aprendemos como assinar um aplicativo e como é importante guardar o par de chaves com segurança. Olhamos com mais

atenção aos detalhes para publicar a nossa aplicação no Google Play Console, e como é preciso preencher corretamente os dados solicitados nas abas obrigatórias.

No próximo capítulo, falaremos exclusivamente da plataforma iOS, sobre como emular, assinar e publicar.

Capítulo 6
Detalhes exclusivos da plataforma iOS

A plataforma iOS pode ser bem confusa até mesmo para quem já publicou um aplicativo. A seguir, veremos um panorama do fluxo de desenvolvimento que seguiremos – algo que gostaria de ter visto quando comecei.

Figura 6.1: Panorama de desenvolvimento iOS

Resumindo a figura anterior, com um *Apple ID*, acessamos o site de desenvolvimento `Developer Apple` **(1)** e, dentro dele, geramos todos os arquivos necessários para identificar e validar a aplicação. Em seguida, no `Xcode` **(2)**, podemos logar com o Apple ID de desenvolvedor, assinar o aplicativo – seja para teste ou para produção –, e gerar o que chamam de *build*.

É esse pacote que será enviado posteriormente para o `iTunes Connect` **(3)**, que é o site onde inserimos uma nova aplicação e preenchemos todos os dados (informações, preço, print das telas) para publicar na App Store. Veremos passo a passo cada um desses pontos para podermos publicar o Freeburguer.

6.1 Apple Developer Account

Para ter uma conta de desenvolvedor do tipo desenvolvedor (individual ou empresarial), é necessário pagar anualmente uma taxa de 99 dólares. Caso você opte por registrar sua empresa, será necessário solicitar um **DUNS NUMBER**. Esse número é um código internacional que identifica a empresa.

Os passos para conseguir esse número são:

- Primeiro, você deve criar uma Apple ID (se não tiver), que é uma conta na Apple. Isso é de graça, basta acessar: https://appleid.apple.com/account#!&page=create.
- Depois, você deve solicitar o DUNS NUMBER no endereço: https://developer.apple.com/enroll/duns-lookup/#/search.

Após receber acesso, você poderá entrar em https://developer.apple.com. Além das informações da sua conta, é dentro deste painel administrativo que você gerará e gerenciará os seguintes itens:

Recurso	Para que serve?
Certificates	São certificados gerados para identificar quem está publicando o app.
Keys	Diferente de certificados, as chaves não expiram e são usadas também para identificação.
Identifiers	Identifica um novo app e quais os serviços da Apple ele usa.
Devices	Registra dispositivos para uso em testes.
Provisioning Profiles	Em união com um *Identifier* e um *Certificate*, esse arquivo é necessário para conseguirmos submeter nosso app para a App Store.

Em resumo, *Certificates*, *Identifiers* e *Provisioning Profiles* são usados em união para identificar e validar que o aplicativo é de um desenvolvedor autorizado, e que está vindo diretamente da App Store.

Agora, **abra o Xcode**, navegue até `Xcode -> Preferences -> Accounts` e **adicione sua conta** de desenvolvedor (Apple ID). Assim, poderemos assinar e validar o build futuramente.

6.2 Criando um certificado

Para gerar um certificado manualmente, você vai precisar do *Certificate Signing Request* (CSR). Para criar, acesse em seu Mac o aplicativo `Acesso às chaves` (ou `Keychain Access`).

Dentro do aplicativo, naveque em `Acesso às chaves -> Assistente de Certificado -> Solicitar um Certificado de uma Autoridade de Certificação...`.

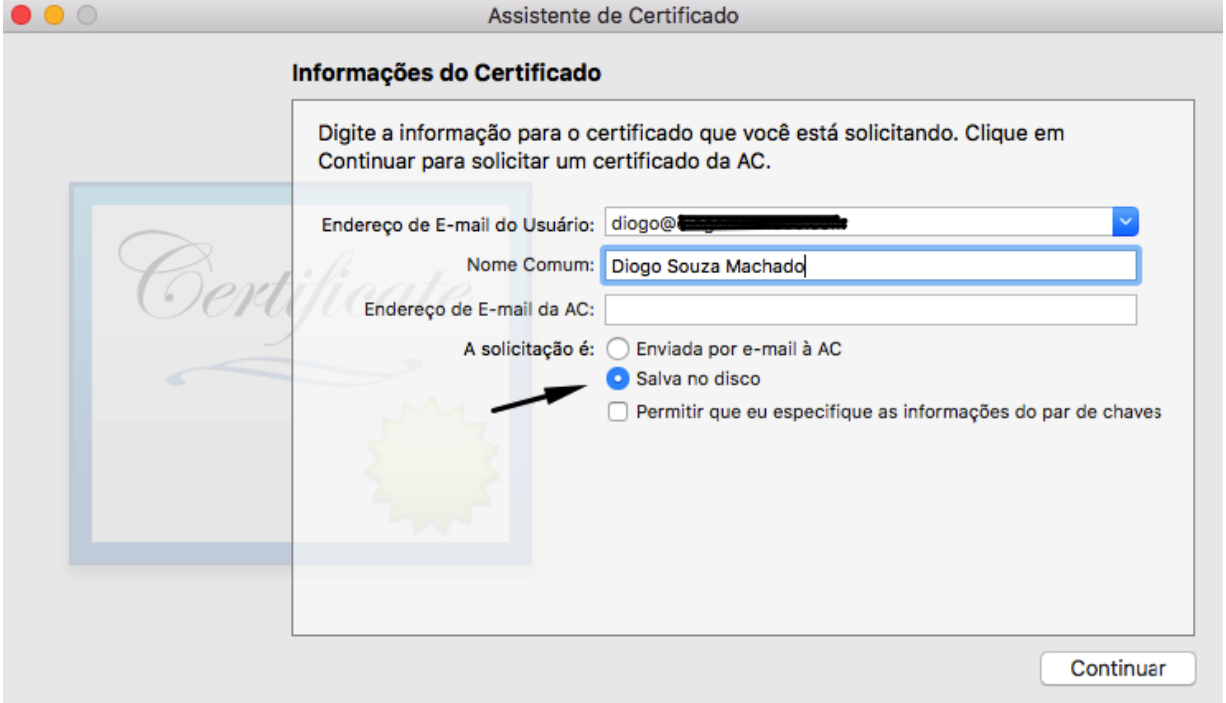

Assistente de Certificado
Informações do Certificado
Digite a informação para o certificado que você está solicitando. Clique em Continuar para solicitar um certificado da AC.
Endereço de E-mail do Usuário:
diogo@
Nome Comum:
Diogo Souza Machado
Endereço de E-mail da AC:
A solicitação é:
Enviada por e-mail à AC
Salva no disco
Permitir que eu especifique as informações do par de chaves
Continuar

Figura 6.2: Janela para preencher informações

O certificado CSR gerado cria uma **chave pública** e uma **chave privada**. Em seu Mac, a chave privada é armazenada e pode ser visualizada dentro do aplicativo `Acesso às chaves` e encontrada na categoria `Chaves`. A parte pública da chave é o arquivo salvo no seu computador, que você vai enviar ao site da Apple.

Agora, dentro do site https://developer.apple.com, você deve criar um novo certificado e fazer o upload do arquivo CSR salvo em sua máquina. Após gerar o certificado, faça o download e clique duas vezes no arquivo para abri-lo. Assim, o computador conseguirá fazer a importação para o software de chaves.

6.3 Registrando um App ID (Identifier)

Diferente dos certificados gerados de tempos em tempos, um *identifier* deve ser criado a cada aplicativo publicado por você. A

App ID String, que identifica seu aplicativo, será gerada com duas informações: o **ID Prefix**, que identifica o *Team ID*, um número gerado pela Apple (algo como *3PORFY95B6*); e o **ID Suffix**, escolhido por nós.

Existem dois tipos de *ID Suffix* para um aplicativo:

- **Explicit App ID**: deve conter um nome único; a Apple recomenda usar o método "string no estilo nome de domínio reverso". Lá no arquivo `config.xml`, usamos o seguinte nome `io.apps.freeburguer`, e vamos usar o mesmo aqui.
- **Wildcard App ID**: serve para caso se queira usar o mesmo ID para várias aplicações. Ele é ideal para aplicações que têm ligação entre elas, como um login universal. Porém, é importante lembrar que, com ele, não é possível usar os serviços da Apple como o *Apple Pay*, *Push Notifications* e outros.

Somados ID Prefix e ID Suffix, temos uma identificação única para o aplicativo, algo como `3PORFY95B6.io.apps.freeburguer`.

Depois de criar o App ID, você está habilitado a criar *Provisioning Profiles*, que será o nosso próximo assunto.

6.4 Gerando o perfil (Provisioning Profile)

Em resumo, um *Provisioning Profile (Perfil)* autoriza a instalação do aplicativo nos dispositivos iOS. Um perfil é a união de seu certificado mais a identificação do dispositivo (no caso de testes) e o App ID.

Development provisioning são usados para gerar e instalar versões do seu app durante o ciclo de desenvolvimento, e os *Distribution provisioning* são usados para submeter a aplicação para a App Store.

A principal diferença de um perfil de desenvolvimento para o de distribuição é que, no de desenvolvimento, você deve selecionar os dispositivos autorizados; já no de distribuição, não é necessário.

Gerando manualmente pelo site de desenvolvedor, em https://developer.apple.com, você vai selecionar o App ID e o certificado (aquele gerado no computador). Então, você poderá baixar o arquivo e abri-lo no seu Mac.

Xcode gera perfis automaticamente

Apesar de termos como gerar manualmente, o Xcode possui uma opção dentro das configurações do app (em `General -> Signing`) para gerar os perfis automaticamente.

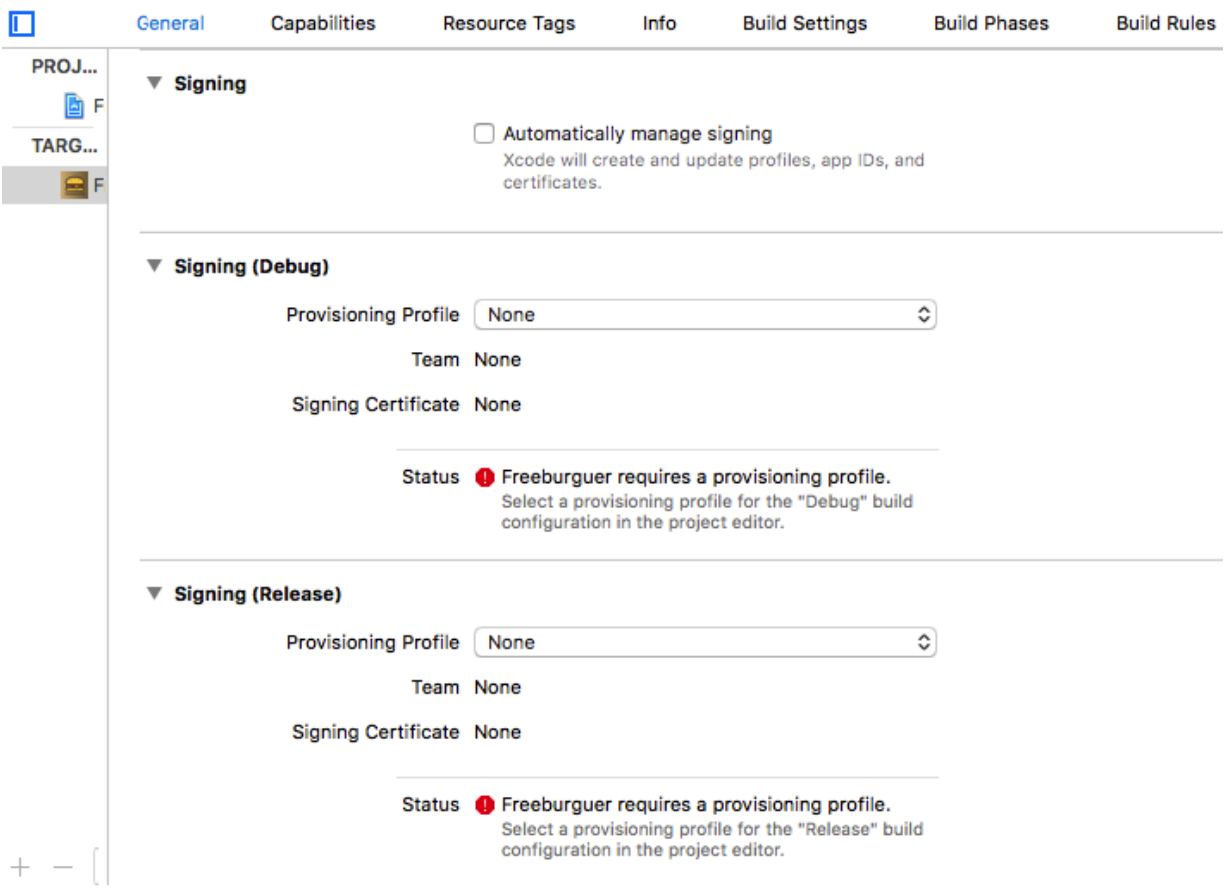


Figura 6.3: Opção de geração automática dentro do Xcode

6.5 Emulando a aplicação no Xcode

Para emular o iOS, você não tem para onde fugir. Será preciso usar o Xcode. No capítulo *Configurando seu ambiente de desenvolvimento*, na seção *iOS no MacOS*, já baixamos todos os emuladores previamente. Esse processo demora um pouco, então, se não fez isso, sugiro que volte ao capítulo e faça.

Quando adicionamos a plataforma do iOS, o Cordova adicionará um arquivo projeto do Xcode. Ele encontra-se no caminho `freeburguer -> platforms -> ios -> Freeburguer.Xcodeproj` . É só dar um clique duplo para o Xcode ser aberto.

Para emular, basta escolher uma opção de dispositivos, como na figura a seguir, e pressionar o botão de *play* para inicializar o emulador:

Figura 6.4: Seleção de dispositivo para emular

Após pressionar o botão, o projeto será gerado, o emulador inicializado, e o aplicativo será aberto automaticamente no dispositivo virtual.

Existe uma área no Xcode que pode ser útil para nós, localizada na IDE na área central, mais à direita. Lá o software nos dá todo o **log** de inicialização do aplicativo. Esse log é diferente da inspeção do projeto, feita no Safari (como vimos no capítulo *Desenvolvendo as funcionalidades do aplicativo*), e pode nos ajudar quando estivermos tentando usar algum plugin ou mesmo uma configuração mais específica.

Figura 6.5: Log do Xcode

6.6 Empacotando o aplicativo e enviando o Build

Diferente do Android, o Xcode já faz o processo de assinatura automaticamente. Basta configurar os certificados corretamente. Feito isso, a própria IDE assinará digitalmente.

Para publicar um aplicativo na Apple Store, é necessário fazer o envio de um **build** no Xcode. Para gerar o build do Freeburguer, abra o arquivo `Freeburguer.Xcodeproj` .

> Daqui em diante, acredito que você já esteja com sua conta logada no Xcode. Caso não tenha feito, volte à seção *Apple Developer Account* do capítulo.

Selecionando um modo de assinatura

Já sabemos criar um perfil *Provisioning Profile* de forma manual, mas acontece que o Xcode nos dá a opção de fazer isso automaticamente. Essa escolha fica por sua conta, e a opção para isso está dentro do projeto no Xcode na aba `General` , como na figura a seguir.

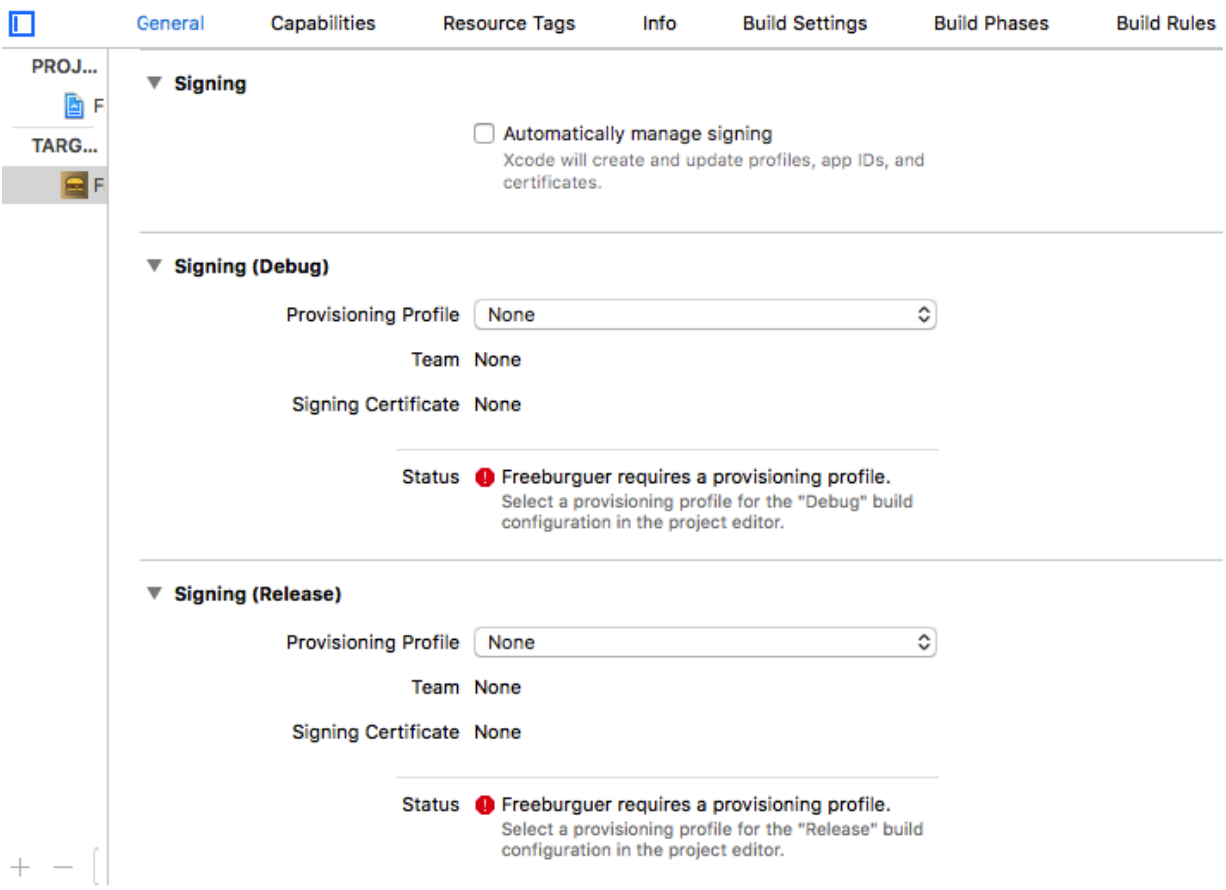


Figura 6.6: Opções de assinatura no Xcode

O próximo passo agora é selecionar o dispositivo `Generic iOS Device`. Essa seleção ocorre no mesmo local onde selecionamos outros dispositivos para simulação.

Agora, acesse o menu `Product -> Archive`. Essa opção vai compilar o projeto e gerar o build.

Se a compilação ocorreu corretamente, uma janela se abrirá para a etapa final de envio. Mas caso tenha ocorrido algum erro, o Xcode lhe informará. Em geral, o erro mais comum é em relação à assinatura. Basta revisar os passos da sua conta e da geração do perfil. Em relação à janela que se abrirá, veja a figura a seguir.

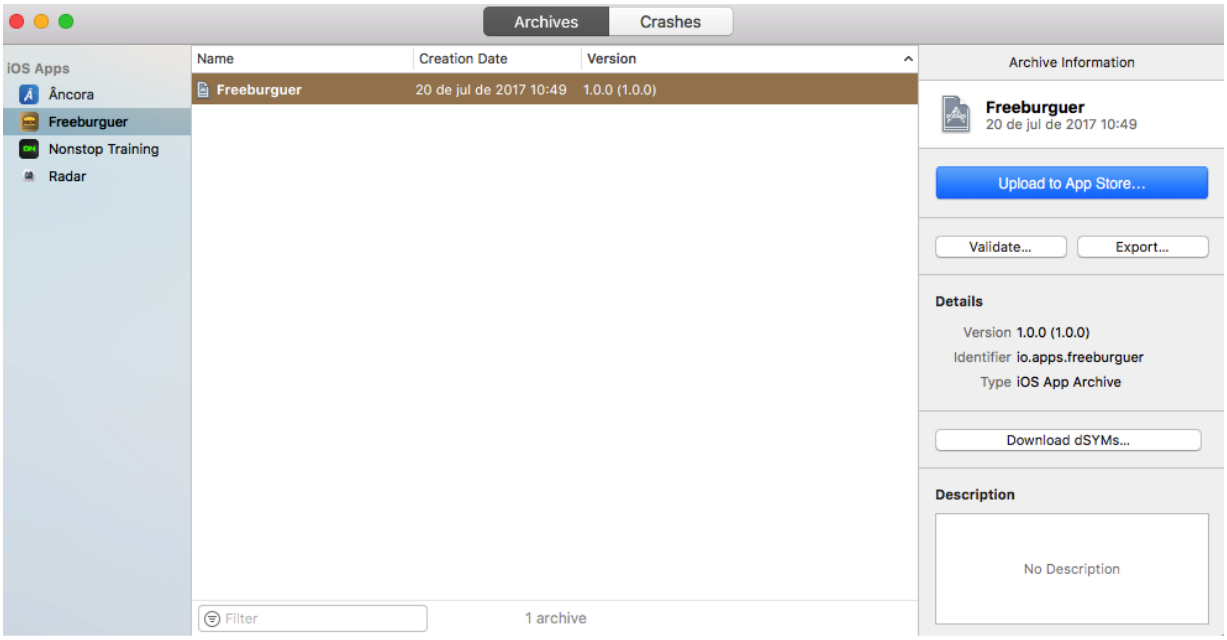


Figura 6.7: Janela de finalização do build

Neste momento, você deve acessar o https://itunesconnect.apple.com para criar um novo aplicativo. **Isso é importante**, porque senão você não conseguirá fazer o upload para a App Store. Dentro do site, vá em `My Apps -> Ícone mais (+) -> New App`.

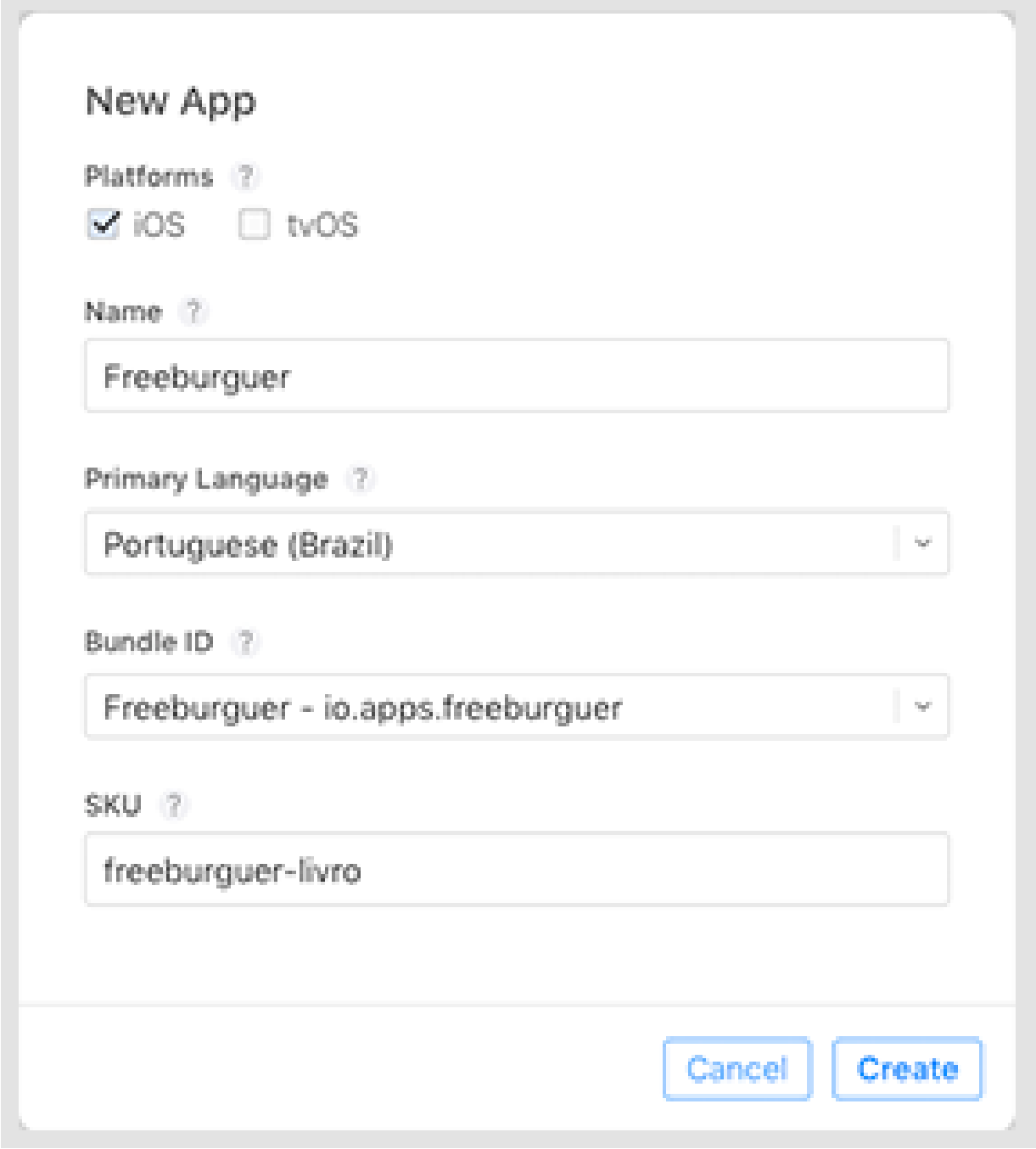


Figura 6.8: Modal de informações do novo app

Pronto, voltando ao Xcode na janela aberta, após o comando `archive` (esta chama-se **Organizer** e pode ser reaberta em `Window -> Organizer`), vamos clicar em `Validate...`. Esse comando vai validar se o pacote compilado está correto e, após a validação, se tudo estiver certo, clicaremos em `Upload to App Store`.

Após o envio do build, ele não será disponibilizado imediatamente dentro do iTunes Connect. A Apple envia um e-mail quando ele estiver disponível para seleção no painel do iTunes Connect, dentro das informações do seu aplicativo.

6.7 Publicando o aplicativo usando o iTunes Connect

Basicamente, essa etapa é preencher informações sobre o seu aplicativo. Com o tempo, algumas coisas podem mudar, então é importante ficar atento a alguns pontos que considero importante nesta etapa.

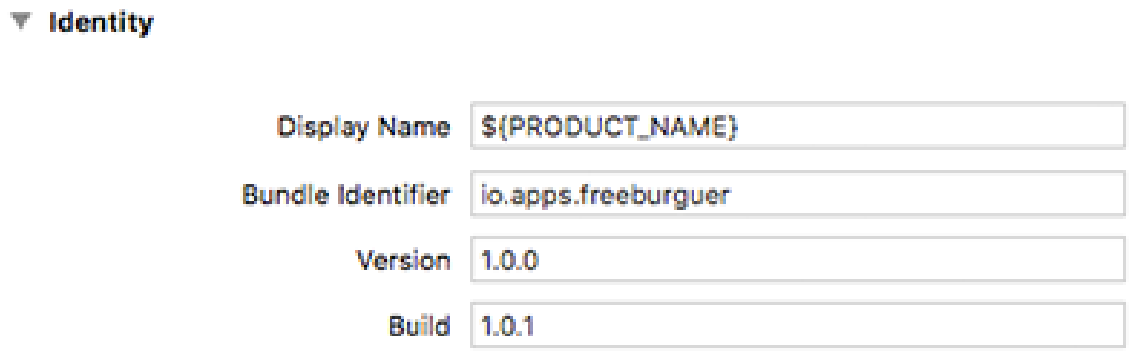
- **Ícone quadrado**: não se esqueça de enviar o ícone do aplicativo quadrado, sem bordas. O próprio sistema vai cortá-lo para exibir na App Store.
- **Version Release**: se você marcar *Manually release this version* (lançamento manual), após a avaliação do aplicativo, ele só será lançado na loja se você clicar no botão para publicar.
- **App Review Information**: se o seu aplicativo tem áreas restritas com login e senha, os avaliadores da Apple vão querer acessar para ver como funciona. Você deve passar todas as informações para que eles tenham acesso.

> **GERANDO OS SCREENSHOTS**
>
> Uma forma de gerar os screenshots (aquelas telas do seu aplicativo) é abrir o simulador no Xcode e apertar `Command + S`. A Apple tem uma especificação sobre os tamanhos e regras para screenshots, e você pode ver a tabela em http://apple.co/2uhQmPw.

6.8 Saiba como lançar uma nova versão do aplicativo iOS

Na hora de lançar uma nova versão, a primeira coisa a se fazer é alterar o arquivo `config.xml` . Em `version=x.x.x` , você vai atribuir uma nova versão. Sugiro que siga as práticas do versionamento semântico.

> **VERSIONAMENTO SEMÂNTICO**
>
> São um conjunto de regras simples que ditam como os números de versões são atribuídos e incrementados. Saiba mais em: http://semver.org/lang/pt-BR.

> **VERSIONANDO O BUILD**
>
> No iOS, é possível versionar o build explicitamente no `config.xml` com o atributo `ios-CFBundleVersion='x.x.x'` , na tag `<widget>` . Feito isso, ao abrir o projeto no Xcode, você verá o seu versionamento diferente da versão, como na figura a seguir.
>
> 
>

Agora, depois de aumentar a versão, gerar o Build do seu aplicativo no Xcode e enviar para o iTunes Connect, vá em `My Apps -> Freeburguer -> + Version or Platform -> iOS` , dentro do iTunes Connect,

e insira o número da versão exatamente como escolhemos no `config.xml`.

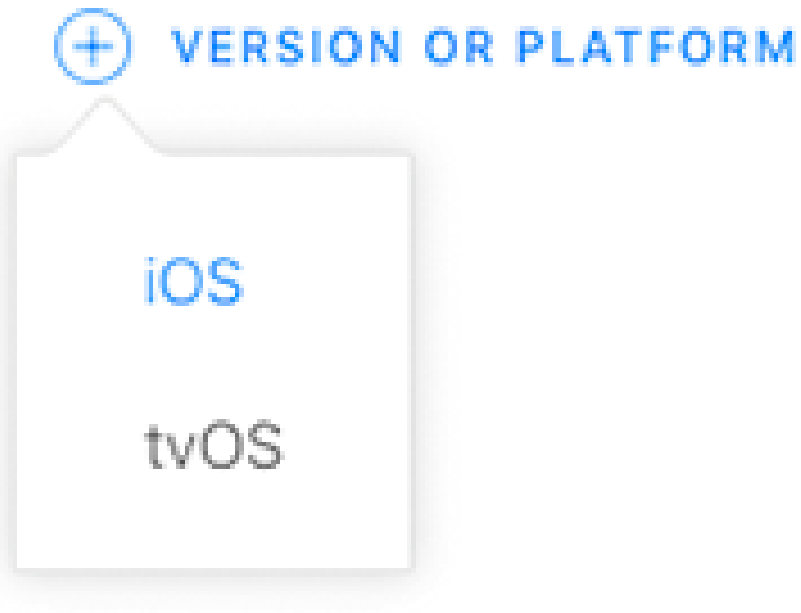


Figura 6.10: Botão na iTunes Connect

Com o novo build enviado e a versão adicionada ao app, você precisa **preencher as informações da versão**. Nesse momento, você pode alterar os *screenshots* e outras informações pertinentes da versão.

6.9 Melhorando a performance com WKWebView

Por padrão, o iOS trabalha com uma WebView chamada **UIWebView**, porém existe uma opção mais rápida que é o **WKWebView**. Ele traz consigo grandes vantagens em relação ao UIWebView, como o consumo reduzido de memória, melhor desempenho de renderização, suporte à funcionalidade *indexedDB*, eventos de rolagem confiáveis e grande estabilidade. Ele também é atualizado junto com os lançamentos do sistema iOS.

Para instalar esse plugin, é bastante simples: basta rodar o comando padrão para adicionar plugins! Repare que estamos adicionando o plugin do repositório do Ionic Team. Escolhemos esse repositório porque a equipe do Ionic fez algumas melhorias e correções importantes, e pensam em fazer um *pull request* delas no futuro para o repositório oficial do WKWebView.

> Mais informações sobre o WKWebView, vale a pena ler o artigo feito pela equipe do Ionic: http://bit.ly/2GhRsyN.

Execute o comando a seguir:

```
cordova plugin add cordova-plugin-ionic-webview --save
```

É importante também adicionar as permissões a seguir no arquivo `config.xml`:

```
<allow-navigation href="http://localhost:8080/*"/>
<feature name="CDVWKWebViewEngine">
  <param name="ios-package" value="CDVWKWebViewEngine" />
</feature>

<preference name="CordovaWebViewEngine" value="CDVWKWebViewEngine" />
```

Pronto! Para verificar se a WebView está funcionando, podemos ver se a funcionalidade `indexedDB` está disponível usando a inspeção do aplicativo com o *Safari*. Assim, sabemos que, por padrão (no UIWebView), essa funcionalidade não está disponível.

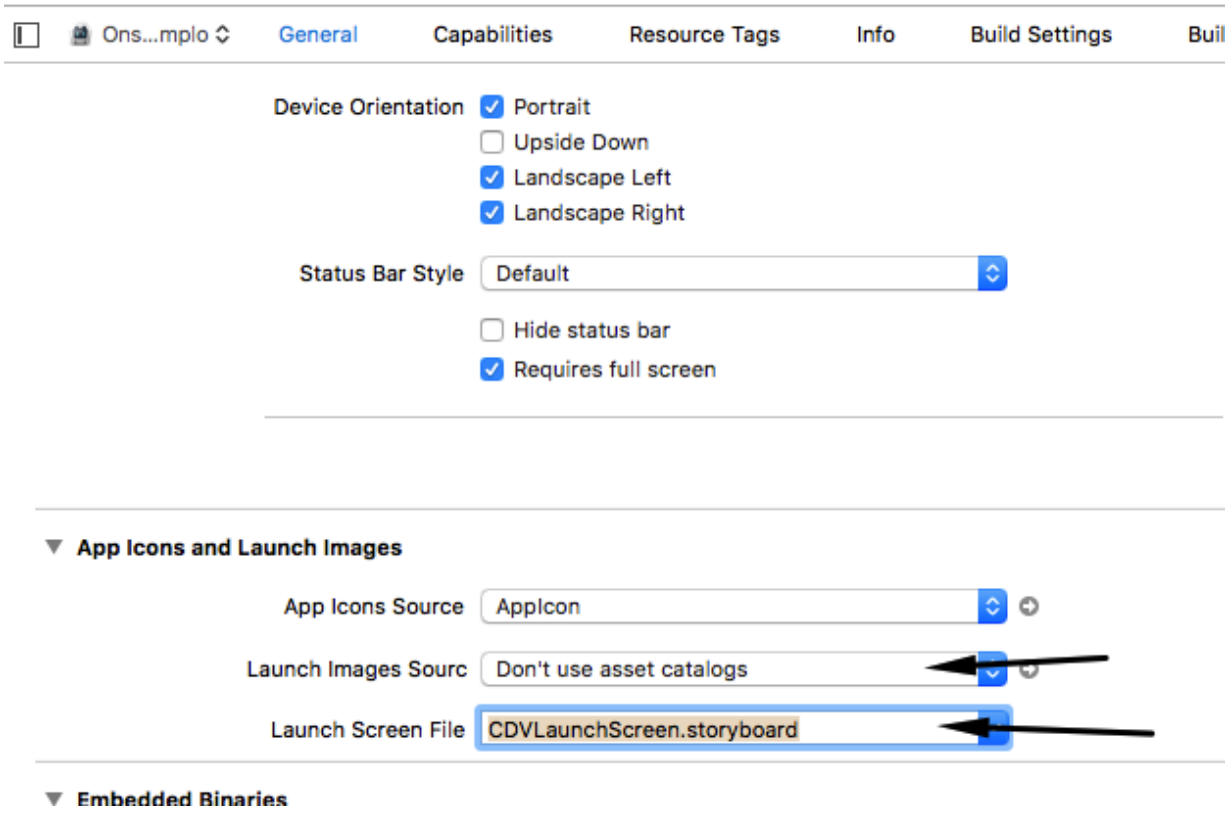
```
if (window.indexedDB) {
   console.log("WKWebView ativada!");
} else {
   console.log("UIWebView ativada");
}
```

6.10 Como desativar o Launch Images no iOS

A plataforma iOS possui um recurso chamado *Launch Images*, que é basicamente o splashscreen nativo deles. Logo, se você não quiser usar splashscreens em seu projeto Cordova para iOS, ainda assim você verá uma imagem aparecer logo ao abrir o aplicativo.

Para desativar as Launch Images, você deve abrir o projeto do seu aplicativo no Xcode. Lembre-se de que o projeto só estará disponível após você adicionar a plataforma com `cordova plataforma add ios`. O arquivo do projeto está localizado dentro do diretório `platforms -> ios`, com a extensão de `.xcodeproj`.

Após abrir o Xcode, clique em `Gereral`, em `App Icon and Launch Images`, vá até `Launch Images Sourc` e selecione `Don't use asset catalogs`. Logo abaixo, em `Launch Screen File`, selecione `CDVLaunchScreen.storyboard`.



Essas configurações vão desativar as *Launch Images* e o aplicativo abrirá direto.

Você pode baixar o aplicativo pronto até aqui no link http://bit.ly/cordova-livro. O projeto é o **freeburguer-cap6**.

6.11 Revisão

Neste capítulo, conhecemos todo o processo para publicação do aplicativo na App Store. Percebemos como a Apple possui um processo bem criterioso para identificar tanto os seus desenvolvedores quanto os aplicativos que estão ingressando na loja. Por isso, existe a necessidade de criar certificados, App IDs e perfis. Também explicamos como modificar o WebView padrão do iOS para o WKWebView, e assim obter ganhos significativos de performance e funcionalidades.

Até aqui, falamos de todos os aspectos necessários para produzir aplicativos incríveis usando o Cordova e seus plugins, em união a um framework SPA (no nosso caso, usamos o AngularJS). Aprendemos sobre design, aspectos específicos de cada plataforma e, o mais importante, como publicá-los!

Mas não pense que acabou. Separei o último capítulo com um conteúdo especial para fechar com *chave de ouro* a produção do aplicativo Freeburguer. Assim, você pode aprimorar ainda mais o seu conhecimento no mundo híbrido de apps.

Capítulo 7
Hooks, armazenamento local, segurança e muito mais!

Neste capítulo, resolvi falar dos detalhes. A ideia é mostrar ferramentas que ajudam a aprimorar a entrega final de um aplicativo híbrido, recursos para *minificar* arquivos CSS, como realizar configurações de segurança e como escolher a melhor alternativa de armazenamento são alguns exemplos do que deixo para vocês complementarem seus estudos.

7.1 Minificar e concatenar arquivos CSS e JavaScript

Quando estamos construindo um novo aplicativo, dependendo da sua forma de organizar os arquivos de estilo CSS, você terá ou um arquivo muito grande, ou vários arquivos quebrados em componentes e outras coisas que desejar – fora os CSS de plugins que baixar e jogar no projeto. O fato é que, como o nosso aplicativo estará rodando dentro de um navegador (o famoso WebView), fazer muitas requisições e ter arquivos enormes pode atrapalhar o tempo que ele leva para interpretar a folha de estilo e *pintar* esses componentes na tela.

Uma prática já antiga de quem trabalha com desenvolvimento web é **minificar** todo o código que vai para produção. Uma ferramenta de compilação bastante conhecida é o Gulp, que executa tarefas, é escrito em NodeJS e pode ser facilmente instalado via *npm*, conforme instalamos outros pacotes como o do Cordova.

A grande vantagem de usar o Gulp é que ele agrega uma série de plugins, como o de minificar código CSS, o de minificar JavaScript e

assim por diante – a ferramenta na qual escrevo este livro conta com 3.349 plugins. Outra vantagem de usar o Gulp é a facilidade de aprender como as coisas funcionam. A desvantagem seria você ter de aprender mais um assunto, mas não se engane; vale a pena aprender a usá-lo.

Partindo para instalar o Gulp, com o terminal aberto no diretório do projeto, rodamos o comando a seguir:

```
npm install gulp-cli -g
npm install gulp -D
```

Essa ferramenta trabalha como um automatizador de tarefas e vistoria um arquivo chamado `gulpfile.js` . É nele que vamos escrever os comandos de minificação. Crie o arquivo `gulpfile.js` dentro da raiz do projeto Freeburguer. Nesse arquivo, escreveremos o código em NodeJS para minificar as folhas de estilo do nosso app.

O primeiro passo de código é criar um array para coletar os caminhos dos arquivos CSS que queremos concatenar e minificar:

```
var css = [
    './www/assets/css/*'
];
```

Em seguida, inicializamos as dependências usando `require` – cada dependência terá um papel no código:

```
var gulp             = require('gulp');
var uglify           = require("gulp-uglify");
var concat           = require("gulp-concat");
var cssmin           = require("gulp-cssmin");
var stripCssComments = require('gulp-strip-css-comments');
var watch            = require('gulp-watch');
```

Para a tarefa de minificar, precisamos das dependências `gulp-uglify` para minificar JavaScript, `gulp-concat` para concatenar arquivos, `gulp-cssmin` para minificar arquivos CSS, `gulp-strip-css-comments` para remover comentários e `gulp-watch` para ficar observando as mudanças, lembrando de que não basta chamá-las no código: você

precisa tê-las instaladas via **npm**. Sendo assim, com terminal aberto, vá até a raiz do seu aplicativo e execute a instalação com o comando a seguir:

```
npm install gulp-uglify gulp-concat gulp-cssmin gulp-strip-css-comments
gulp-watch
```

Agora, vamos criar uma *tarefa* do Gulp, que precisa de um nome. Vamos chamá-la de `minify-css`. Acompanhe o código a seguir:

```
gulp.task('minify-css', function(){
    gulp.src(css)
    .pipe(concat('app.min.css'))
    .pipe(stripCssComments({all: true}))
    .pipe(cssmin())
    .pipe(gulp.dest('./www/assets/css/'));
});
```

Repare que o código anterior chama `concat`, passando um nome para o arquivo final gerado. Depois, `stripCssComments` remove os comentários do CSS, `cssmin` minifica os estilos e `gulp.dest` informa o destino do arquivo final gerado. Isso é tudo o que precisamos para nossa missão.

Por fim, para que possamos executar essa tarefa, vamos adicioná-la à execução *default* do Gulp. Assim, ao rodar o comando `gulp`, ele já vai executar a tarefa `minify-css`.

```
gulp.task('default',['minify-css']);
```

Abra o terminal no diretório raiz do projeto, onde está o `gulpfile.js` em que estamos trabalhando, e execute o comando a seguir:

```
gulp
```

O resultado esperado será o arquivo final `app.min.css`, gerado no diretório que apontamos `./www/assets/css/`.

Quando trabalhamos com Gulp, pode ser cansativo ficar executando o comando `gulp` no terminal a cada alteração. Por isso, podemos

usar a dependência `gulp-watch` para que o próprio Gulp faça a execução do comando a cada alteração. Só precisamos adicionar ao arquivo `gulpfile.js` as seguintes linhas:

```
gulp.task('watch', function() {
    gulp.watch(css, ['minify-css']);
});
```

Com o código anterior, sempre que quiser ativar a *escuta* de alterações para o Gulp rodar automaticamente, abra o terminal no diretório (onde se encontra o `gulpfile.js`) e execute o comando:

```
gulp watch
```

Com o arquivo gerado, podemos ter apenas uma chamada para `app.min.css`, **economizando requests** e com um arquivo **mais leve**. Confira o arquivo `gulpfile.js` completo a seguir:

```
var css = [
    './www/assets/css/*'
];

var gulp             = require('gulp');
var uglify           = require("gulp-uglify");
var concat           = require("gulp-concat");
var cssmin           = require("gulp-cssmin");
var stripCssComments = require('gulp-strip-css-comments');
var watch            = require('gulp-watch');

gulp.task('minify-css', function(){
    gulp.src(css)
    .pipe(concat('app.min.css'))
    .pipe(stripCssComments({all: true}))
    .pipe(cssmin())
    .pipe(gulp.dest('./www/assets/css/'));
});

gulp.task('default',['minify-css']);

gulp.task('watch', function() {
```

```
    gulp.watch(css, ['minify-css']);
});
```

7.2 Usando hooks para automatizar o Cordova

O conceito de **hooks**, algo como "*um gancho para alguma coisa que vai acontecer*", está presente em várias ferramentas de software, e um bem conhecido é o Git. O Cordova adotou esse conceito para que possamos ter mais uma opção de customização de tarefas após ou antes de executar algum evento, como o `cordova build`.

Imagine que, por algum motivo, o desenvolvedor deseja saber o tamanho do `apk` gerado na plataforma Android sempre após rodar o comando de *build* do Cordova. O problema é que isso é algo bem específico, mas que pode ser resolvido com um script *hook*.

Para trabalharmos com hooks no Cordova, temos na raiz da aplicação um diretório chamado `/hook`. Como já dito, essa funcionalidade é utilizada para automatizar tarefas após serem disparados **gatilhos de eventos** do Cordova – como no processo de *build*, quando rodamos o comando `cordova build` no terminal.

Vamos criar o hook dentro do seu diretório, chamando o arquivo de `after_build.js`. Ele será escrito em NodeJS. O principal motivo de usar o NodeJS é que ele é multiplataforma, portanto, esse mesmo script rodará normalmente em uma máquina Windows, Linux ou MacOS, mantendo a ideia principal do framework ser multiplataforma.

Seguindo a documentação oficial (https://cordova.apache.org/docs/en/latest/guide/appdev/hooks/), quando estamos escrevendo um hook com NodeJS, começamos por empacotar toda a lógica dentro de um `module.exports`:

```
module.exports = function(context) {

}
```

A próxima linha do nosso script vai fazer a união do caminho raiz do projeto ao restante gerado pelo Cordova. Para formar o caminho completo da plataforma Android dentro do projeto Cordova, faremos:

```
var platformRoot = path.join(context.opts.projectRoot,
'platforms/android');
```

Em seguida, com o `platformRoot`, vamos criar uma segunda variável, que é o caminho exato do `apk` gerado após o *build*:

```
var apkFileLocation = path.join(platformRoot, 'build/outputs/apk/android-
debug.apk');
```

Para finalizar, utilizando `fs`, verificamos se tudo ocorreu bem. Se tivermos um erro, rejeitamos a promessa com um `reject()`; caso contrário, resolvemos a promessa e também imprimimos o *log* no terminal com o `console.log`.

```
fs.stat(apkFileLocation, function(err,stats){
    if (err){
        deferral.reject('Erro na operação');
    }else{
        console.log('O APK ' + apkFileLocation + ' tem ' + stats.size +'
bytes de tamanho');
        deferral.resolve();
    }
});
```

A última linha do nosso código embutido em `module.exports` é retornar essa promessa criada lá na declaração de variáveis:

```
return deferral.promise;
```

Agora, vamos declarar o hook para que esse script seja executado sempre após realizar o comando `build`. Assim, no arquivo `config.xml`, inserimos uma declaração hook:

```
<platform name="android">
    <hook src="hooks/after_build.js" type="after_build" />
</platform>
```

Repare que encapsulamos o hook dentro apenas da plataforma Android nas configurações, que é de nosso interesse para esse momento. Já as demais plataformas não precisam verificar esse hook. Para testar, abra o terminal na raiz do aplicativo e execute o `build` do projeto:

```
cordova build android
```

Você verá ao final do *build* no seu terminal algo como a figura a seguir:

Figura 7.1: Impressão no terminal via hook

Esse foi apenas um exemplo do que pode ser feito usando hooks. Tudo dependerá bastante do que você precisa e do seu conhecimento em NodeJS. Você encontra o código completo deste hook em: http://bit.ly/2yh6AIn.

> **EXPLORANDO OUTROS EVENTOS**
>
> Existem outros eventos suportados pelos hooks. Confira a lista completa na documentação oficial: http://bit.ly/2gYk1WC.

7.3 Opções de armazenamento local

Veremos agora as ferramentas que temos disponíveis para fazer armazenamento local (offline). É importante conhecer essas possibilidades, pois nem sempre o usuário estará conectado à internet.

localStorage e sessionStorage

Se você trabalha com web, sabe que `localStorage` e `sessionStorage` são duas APIs web que vieram para suprir as deficiências dos antigos *cookies*. Como rodam em cima de uma WebView e são simples de usar, eles se tornaram o modo mais fácil de manter dados localmente em aplicativos híbridos.

A principal diferença dos dois termos é que o `localStorage` vai manter os dados ao fechar o aplicativo, enquanto o `sessionStorage` manterá até que o app seja encerrado. A quantidade total de armazenamento é normalmente em torno de **5 MB** a **10 MB**, suficiente para armazenar strings JSON.

O uso dessa API é bem simples. Para armazenar, basta criar um item com **nome** e **valor**:

```
localStorage.setItem('<NOME>', '<VALOR>');
```

Na prática, o valor preenchido será uma string de JSON, vinda do servidor e transformada de objeto para string com o método `JSON.stringify()`. Também é possível guardar valores comuns, como um simples número ou nome.

O uso do JSON facilita, pois é fácil transformar a string novamente em um objeto JSON:

```
localStorage.setItem('pedidos', JSON.stringify(jsonResponse));
```

Logo em seguida, para recuperar, transformamos a string para objeto com `JSON.parse`:

```
var dados_locais = JSON.parse(localStorage.getItem('pedidos');
```

A sintaxe para o uso de `sessionStorage` é a mesma: `setItem` para gravar, e `getItem` para coletar de volta. Como já disse, utilize `sessionStorage` para dados que não precisam ser lembrados após fechar o aplicativo, e `localStorage` para dados que precisam ficar armazenados mesmo após o fechar.

IndexedDB

O IndexedDB é uma API que permite armazenar dados localmente no navegador do usuário. Ele combina a possibilidade de se armazenar objetos com o recurso de criar índices, o que torna seu sistema de busca mais robusto comparado a outros métodos.

O IndexedDB não é suportado em alguns WebViews. Entretanto, com o projeto Crosswalk (para Android) e o WKWebview (para iOS), seu suporte pode ser resolvido para podermos usá-lo.

Apesar de usar essa API não ser complicado, a lógica comparada ao uso do `localStorage` é bem mais complicada. Vamos à explicação.

> Você pode testar os códigos desta seção usando o console do Google Chrome.

O primeiro passo para começar a usar o `indexedDB` é abrir um novo banco de dados. Para isso, usamos o método `open()` :

```
var request = window.indexedDB.open("Freeburguer", 1);
```

O valor passado no segundo parâmetro do método `open()` é a versão da sua base de dados. Isso é útil, pois existe um evento chamado `onupgradeneeded` , que é executado quando temos uma chamada ao método `open()` com uma versão nova.

Como não temos nenhum banco ainda implantado, o evento `onupgradeneeded` será executado. Nele vamos criar e definir as *tabelas* do banco e os índices, e vamos popular a base.

```
request.onupgradeneeded = function (event) {

    var db = event.target.result;
    var objectStore = db.createObjectStore("hamburgueres", {keyPath: "id"
});

    objectStore.createIndex("codigo_receita", "codigo_receita", { unique:
true });

    for (var i in estoque) {
        objectStore.add(estoque[i]);
    }
}
```

No código anterior, `createObjectStore()` é usado para criar a tabela `hamburgueres`. O parâmetro `keyPath` é responsável por definir o nome da chave (índice) dentro do objeto que estamos armazenando. No caso, cada hambúrguer da nossa lista terá um ID único criado por nós, como o array a seguir:

```
const estoque = [
    {
        id: "P001",
        codigo_receita: "REC201234",
        nome: "Pão de gergilim"
    }
]
```

ID INCREMENTAL

É possível usar um ID autoincrementado. Veja na documentação do método `createObjectStore()`, em https://mzl.la/2EgDCfm.

Ainda em `onupgradeneeded`, usamos o método `createIndex()`. Intuitivamente, sabemos que esse método é para criar outros índices, como um campo de CPF, por exemplo. No nosso caso, vamos criar um índice do `codigo_receita`, que é outro código único do item da base de dados.

Como é um campo único, também passamos o parâmetro `unique` como `true`. Além disso, é possível criar índices de campos que não são únicos passando `unique` como `false`.

```
objectStore.createIndex("codigo_receita", "codigo_receita", { unique: true
});
```

Para fechar o código dentro do evento `onupgradeneeded`, depois de definir a *tabela* e os índices, vamos popular a base com a constante `estoque`, que possui uma lista de hambúrgueres. Então, o código vai iterando no loop e adicionando a base do IndexedDB:

```
for (var i in estoque) {
    objectStore.add(estoque[i]);
}
```

Hora de testar

Você pode visualizar a base `indexedDB` nas ferramentas de desenvolvedor. No Google Chrome, ela se encontra na aba de `Application -> Storage -> IndexedDB`.

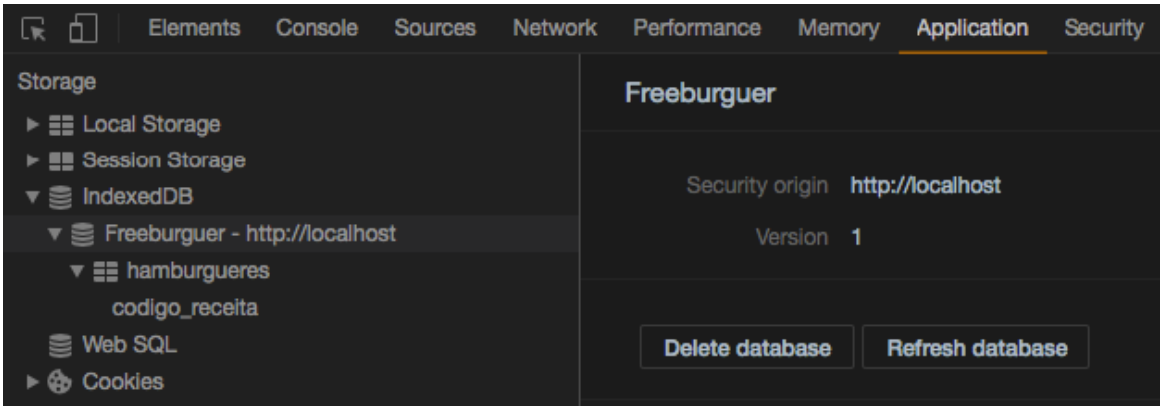


Figura 7.2: indexedDB Freeburguer no Google Chrome

É importante saber agora que toda manipulação em uma base `indexedDB` é feita por transações com o objeto `transaction`. Você

pode testar um exemplo completo usando `indexedDB` no repositório http://bit.ly/cordova-livro, em `indexed-db`.

Qual é o espaço de armazenamento total do IndexedDB?

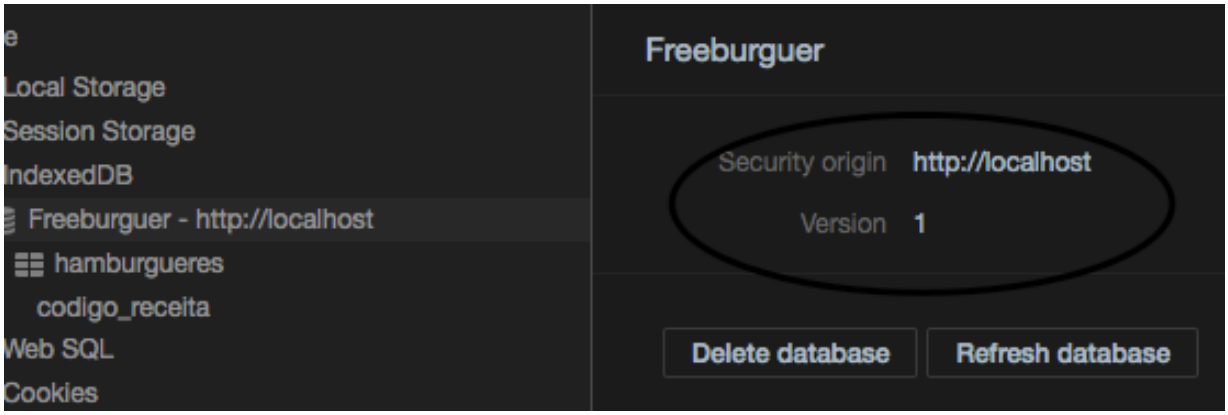
Essa é outra vantagem da API. O espaço total de armazenamento do `indexedDB` depende da regra de cada navegador, mas, em geral, é maior que o do `localStorage`. Baseado na documentação do Google Chrome, o espaço dessa API utiliza o **armazenamento temporário** do navegador.

Esse tipo de armazenamento é um espaço de disco que serve a todos os aplicativos web usados no navegador, e é compartilhado também para as APIs offline (como o `indexedDB`). Esse armazenamento pode ter até 1/3 do espaço do disco, e o espaço de armazenamento disponível é calculado da seguinte maneira:

```
(espaço disponível + espaço já usado) * .333
```

Certo, conhecemos a teoria, mas já dá para perceber que o espaço total de armazenamento do `indexedDB` é bem maior que os 5 MB disponíveis do `localStorage`. Entretanto, ainda existe um método para calcularmos o espaço que o *host* está usando e o qual ainda temos.

Digo *host*, já que cada banco `indexedDB` está unicamente atrelado a um *host*. Por exemplo, se sua aplicação roda em *http://localhost/index.html*, o *host* atrelado é o *http://localhost** e o banco estará disponível apenas nesse domínio quando você abrir as ferramentas de desenvolvedor. É possível provar isso com a figura a seguir:



Não existe um método comum a todos os navegadores para verificar o tamanho do espaço total. Portanto, é sempre bom testar bastante e tentar não manter uma base local enorme devido a essa falta de transparência dos browsers. O Chrome é o que dá mais transparência, e podemos usar o código a seguir para saber o quanto estamos usando e o quanto está livre. Vale lembrar que, no Safari, isso não funciona.

```
navigator.storage.estimate().then(function(estimate) {
  console.log(estimate.usage);
  console.log(estimate.quota);
});
```

Em resumo, o `indexedDB` fornece uma ferramenta mais robusta para armazenar dados localmente. Apesar de não termos certeza do tamanho total em todos os WebViews, sabemos que ele é bem maior que o disponível para as APIs de `localStorage` e `sessionStorage`, já que elas compartilham do espaço temporário que o WebView disponibiliza.

Uma desvantagem em relação ao `localStorage` é a complexidade do uso que aumenta consideravelmente. Ainda assim, não podemos dizer quem é melhor ou pior, pois depende da necessidade de cada aplicativo e situação. Se você quiser usar um exemplo prático, acesse: http://bit.ly/2z9Kvgc.

Gravando em arquivos locais com File Plugin

Com o plugin `cordova-plugin-file` na nossa aplicação, integramos a implementação da File API, permitindo ler e escrever arquivos (áudio, imagem, texto) dentro do aparelho.

> **File API, o que é?**
>
> Uma interface File que provê informações sobre arquivos e permite ao JavaScript acessar seu conteúdo.

Essa solução é adequada para quem deseja gravar **arquivos grandes** dentro da aplicação, como imagens e músicas, mas se torna menos viável para quem deseja manter bases de dados consultáveis localmente. Neste caso, a melhor escolha seria `localStorage` ou `indexedDB`.

Como sua documentação é bem extensa. Se for do seu interesse, sugiro que a leia no repositório oficial: https://github.com/apache/cordova-plugin-file.

WebSQL

Esse recurso segue a ideia de termos uma base usando SQL tradicional – pena que a API foi **depreciada**, por isso não vale a pena conhecer a fundo. Veja alguns motivos para não usar:

- Não suportado por todas as plataformas Cordova;
- Mais complexo para trabalhar com o `localStorage` ou o `IndexedDB`;
- A API está obsoleta;
- Quantidade total limitada de armazenamento (normalmente, em torno de 5 MB).

Outras formas de armazenamento

Além das formas tradicionais de armazenamento local, é muito comum termos de manter também uma base de dados em um

servidor da nuvem, além dos dados locais no aplicativo. A seguir, listo algumas opções de arquitetura:

Usando Realtime do Firebase

A plataforma Firebase implementa um modelo de base de dados NoSQL em tempo real. Isso ajuda a economizar tempo, pois não será necessário se preocupar com a infraestrutura, já que a atualização dos dados em tempo real é feita de forma assíncrona.

Se fôssemos criar um ambiente assim, teríamos de nos preocupar em configurar um servidor com NodeJS rodando o framework Socket.io, para abrir um WebSocket entre o servidor e a nossa aplicação dentro do navegador. Além disso, precisaríamos instalar um banco de dados NoSQL (como o MongoDB, por exemplo). Por isso a plataforma é tão interessante.

> **O que é WebSocket?**
>
> WebSocket é uma tecnologia que permite a comunicação bidirecional por canais *full-duplex* sobre um único soquete *Transmission Control Protocol* (TCP). Ele é projetado para ser executado em browsers e servidores web que suportem o HTML5.

Algumas outras características são:

- Pagamento sobre demanda (só paga pelo consumo que usa);
- Possui suporte offline usando `localStorage`.

Como vantagens, temos:

- Não se preocupar com infraestrutura, somente com a aplicação;
- Serviço mantido pelo Google com alta disponibilidade;
- Comunidade forte;
- Fácil de integrar a qualquer aplicativo JavaScript, usando a SDK JavaScript deles.

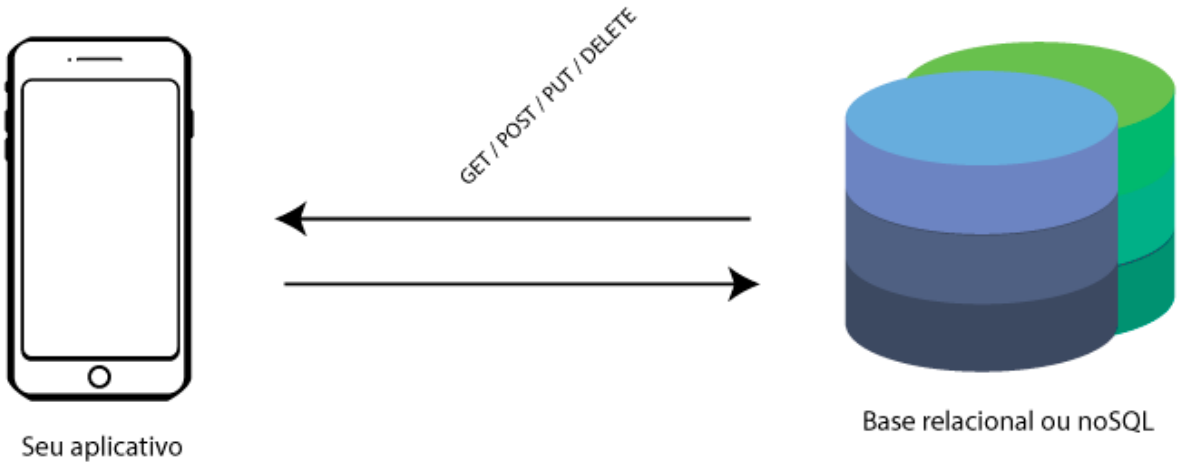
Como desvantagens, temos:

- É pago, então você precisa ficar atento ao tamanho e proporção do projeto que vai implementar.

Explicamos como usar o Firebase no capítulo *Desenvolvendo as funcionalidades do aplicativo*, na seção *Adicionando o Firebase ao projeto*.

Banco de dados externo exposto por API

Uma solução para persistir dados é ter uma aplicação externa que funciona como uma API, construída em qualquer linguagem web, como: o *PHP*, que recebe as chamadas *HTTP* e mantém uma base de dados relacional; o NoSQL, como um MySQL, PostgreSQL; ou MongoDB.



Unida ao uso de `localStorage`, essa lógica é a mais comum para oferecer suporte offline básico. Assim a sua base de dados externa mantém os dados a longo prazo, e o `localStorage` auxilia para manter o necessário para aplicação rodar offline.

Resumindo, você pode programar uma API em qualquer linguagem para web do seu conhecimento – seja Ruby, PHP, Java ou qualquer outra –, expondo essa API em uma URL `https://api.freeburguer.com`, por exemplo. Assim, poderíamos trabalhar algumas chamadas HTTP comuns:

- `GET` – Com JavaScript, fazemos um `GET https://api.freeburguer.com/hamburgueres`, e a API nos retorna a lista em JSON dos hambúrgueres.
- `POST` – Com JavaScript, fazemos um `POST https://api.freeburguer.com/hamburgueres` passando as informações de um novo hambúrguer para a API, que nos retorna um código HTTP `201` que foi adicionado, por exemplo.

Quanto ao banco de dados, você pode usar qualquer um (Oracle, MySQL, MongoDB). O importante é a interface que a API vai entregar para que possamos usar na aplicação JavaScript Cordova.

> **Quer criar APIs para seus aplicativos?**
>
> Se quiser adentrar no mundo das APIs, é importante ler sobre: o modelo REST (*Representational State Transfer*); os códigos que o HTTP possui e como usá-los corretamente; e, claro, dominar uma linguagem de programação back-end. Também vale a pena ler sobre o padrão GraphQL, criado pelo Facebook.

7.4 Dicas de segurança

Em um aplicativo híbrido, é bastante comum usar uma série de ferramentas. Uma delas é o uso de APIs externas e as APIs nativas às quais o Cordova tem acesso. É importante entender alguns conceitos de segurança para se proteger.

- Não use iframes. Prefira o plugin *InAppBrowser*.
- Valide todos os seus inputs.
- Não faça cache de dados sensíveis em `localStorage`. Para isso, use uma base externa (como um MySQL, por exemplo.
- Não considere seu código seguro. É possível fazer engenharia reversa com aplicações Cordova.

A seguir, vamos conhecer um recurso que nos permite controlar o acesso às URLs das quais o seu aplicativo faz chamada no sistema operacional.

Intent whitelist

Este recurso é a forma que temos de o Cordova controlar as URLs que o seu aplicativo pode chamar do sistema operacional. Como o Cordova trabalha dentro de um navegador WebView, era necessário controlar o que podíamos abrir do sistema para evitar aplicativos com intenções maliciosas.

Apesar de esse plugin já vir configurado por padrão, para adicioná-lo, basta rodar o comando no terminal (lembrando de estar na raiz do projeto):

```
cordova plugin cordova-plugin-whitelist --save
```

A configuração é feita no arquivo `config.xml`. Elas aplicam-se a *hiperlinks* e chamadas do comando `window.open()`. A seguir, vamos expor algumas possibilidades de configurações que você pode fazer no seu aplicativo:

Liberar abertura de links em um browser

```
<allow-intent href="http://*/*" />
<allow-intent href="https://*/*" />
```

Caso quiséssemos liberar apenas um domínio específico, poderíamos fazer:

```
<allow-intent href="https://www.alura.com.br/*" />
```

O caractere curinga asterisco (`*`) funciona também para o protocolo:

```
<allow-intent href="*://www.alura.com.br/*" />
```

Liberar abertura do aplicativo de mensagens SMS

```
<allow-intent href="sms:*" />
```

Liberar abertura de telefones direto no aplicativo de chamadas

```
<allow-intent href="tel:*" />
```

Liberar abertura de mapas

```
<allow-intent href="geo:*" />
```

Existem *URLs Intents*, que são exclusivas do Android ou iOS; basta consultar a documentação do Cordova ou do plugin que está configurado. **Evite usar o caractere curinga** por motivos de segurança.

Content Secure Policy

A *Content Secure Policy* é uma política escrita pelo W3C (*World Wide Web Consortium*) que visa criar uma camada extra de proteção contra os ataques de **injeção de conteúdo** principalmente, como o *Cross Site Scripting* (XSS).

> **ATAQUE DE INJEÇÃO DE CONTEÚDO**
>
> É um tipo de ataque no qual o atacante consegue injetar um conteúdo – que pode ser um script malicioso em JavaScript, que envia dados da sua aplicação para outro servidor –, por meio das vulnerabilidades no sistema. Esse tipo de ataque com scripts injetados é conhecido como *Cross Site Scripting* (XSS).

Ao criar um projeto com `cordova create`, ele estará definido no arquivo `index.html` por padrão. Você encontra uma metatag **confusa** definida, como mostra o código a seguir:

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self'
data: gap: https://ssl.gstatic.com 'unsafe-eval'; style-src 'self'
'unsafe-inline'; media-src *; img-src 'self' data: content:;">
```

A principal motivação para entender muito bem essa política é que o WebView não consegue sozinho distinguir os endereços de CSS e JavaScript que são seguros, por exemplo. Por isso, em um caso de injeção de código, um *script* malicioso poderia ser carregado junto sem nenhum problema.

O *Content Secure Policy* fornece ao WebView uma *whitelist* (lista branca) contendo os endereços que são considerados seguros pela aplicação, cujos recursos poderá baixar e usar. Dentro das políticas, temos as *diretivas*, que tratam cada tipo de recurso, como imagens, scripts e folhas de estilo.

As diretivas mais comuns são:

- `default-src` — É um *fallback (retorno)* para outras, ou seja, se nada for definido, o navegador usa as regras da `default-src`.
- `script-src` — Essa política lida com os códigos JavaScript e é a mais notória, já que a maioria dos ataques vem de códigos JS.
- `style-src` — Indica ao navegador como lidar com códigos e arquivos de CSS.
- `img-src` — Indica ao navegador como lidar com as imagens utilizadas pela aplicação.
- `font-src` — Indica ao navegador como lidar com as fontes customizadas usadas pela aplicação.
- `media-src` — Indica ao navegador como lidar com os arquivos de *media* (áudio e vídeo) utilizados pela aplicação.
- `object-src` — Indica ao navegador como lidar com os plugins (flash, PDF etc.) usados pela aplicação.
- `form-action` — Indica ao navegador como lidar com URLs utilizadas em formulários HTML. Nela serão indicados quais endereços podem ser submetidos pelos formulários da aplicação.

Em algumas dessas políticas, além do endereço, é possível definir os seguintes valores: `*`, `'none'`, `'self'` e `'unsafe-inline'`. Ao atribuir o valor curinga (o asterisco) a alguma diretiva, você estará liberando o acesso a todo e qualquer endereço de carregamento de

arquivos daquela diretiva. Evite usar esse curinga em suas diretivas, pois liberar acesso a tudo certamente vai causar vulnerabilidades na aplicação.

Ao atribuir o valor `'none'` a alguma diretiva, você estará bloqueando todo e qualquer endereço de carregamento de arquivos daquela diretiva – inclusive dos arquivos cujo endereço for o mesmo utilizado pela aplicação. Em outras palavras, `none` significa *"bloqueie tudo!"*.

O valor `'unsafe-inline'` pode ser utilizado nas diretivas `script-src` e `style-src`, para permitir a inclusão de códigos CSS e JavaScript inline. Isto é, códigos que não estão em arquivos separados da página, mas sim declarados juntamente com o código HTML.

O valor `'self'` geralmente é o mais usado nas diretivas, pois indica ao browser que ele pode carregar os arquivos cujo endereço seja o mesmo da aplicação. Isso faz com que o browser bloqueie o carregamento de arquivos externos – algo que aumenta bastante a segurança do app.

7.5 Adicionando notificações push

Existem várias maneiras de adicionar suporte a notificações push, e uma das mais fáceis que encontrei foi utilizando os serviços da OneSignal (https://onesignal.com). É um site que provê um serviço multiplataforma de notificações push, sendo o Cordova uma das plataformas compatíveis. E o preço: **de graça**.

> **PRIVACIDADE**
>
> É importante saber que os serviços da OneSignal sem custos compartilham informações da sua conta com terceiros. Essa é a forma que eles ganham dinheiro. Se você deseja enviar dados sensíveis, eles oferecem opções pagas.

Para enviar notificações, independente de você usar o OneSignal ou outro serviço, você precisa das **credenciais do projeto** do Android e do iOS. Veremos como fazer isso a seguir.

Configurando na plataforma Android

Começamos pela plataforma mais fácil, vá até o site do Firebase:

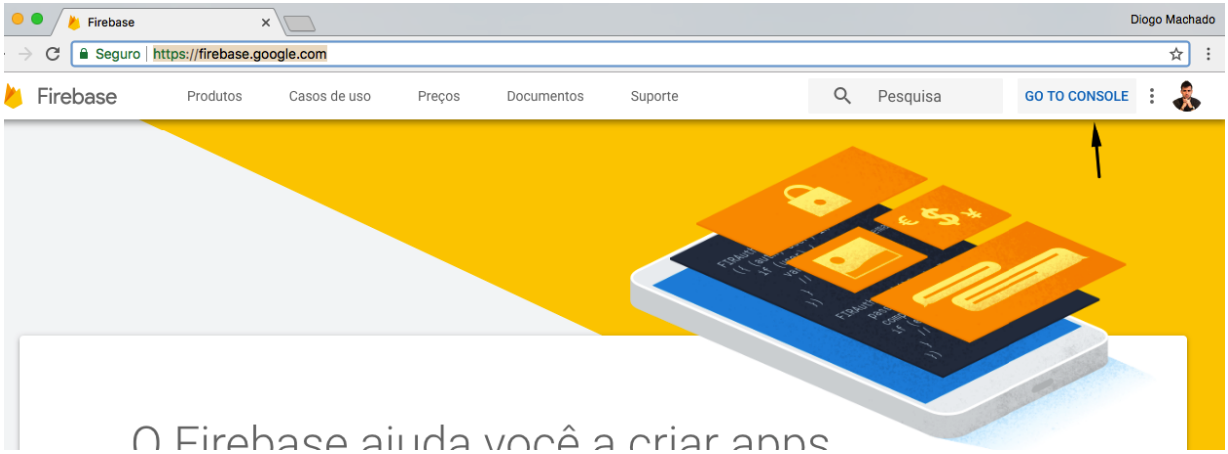
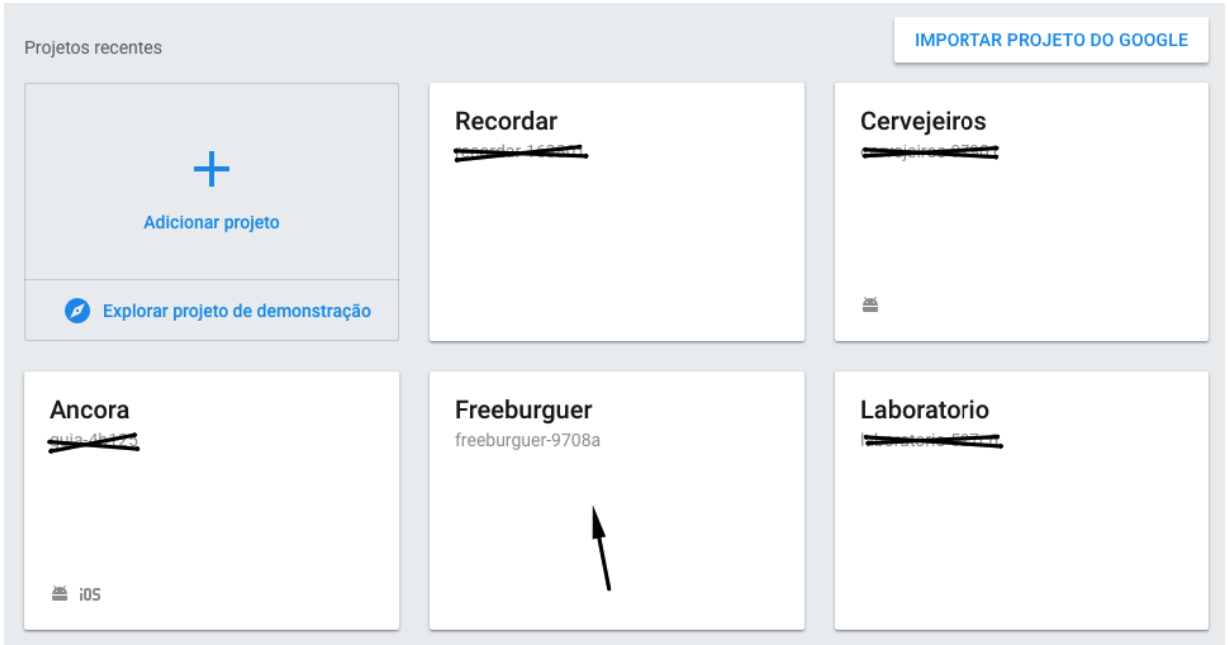
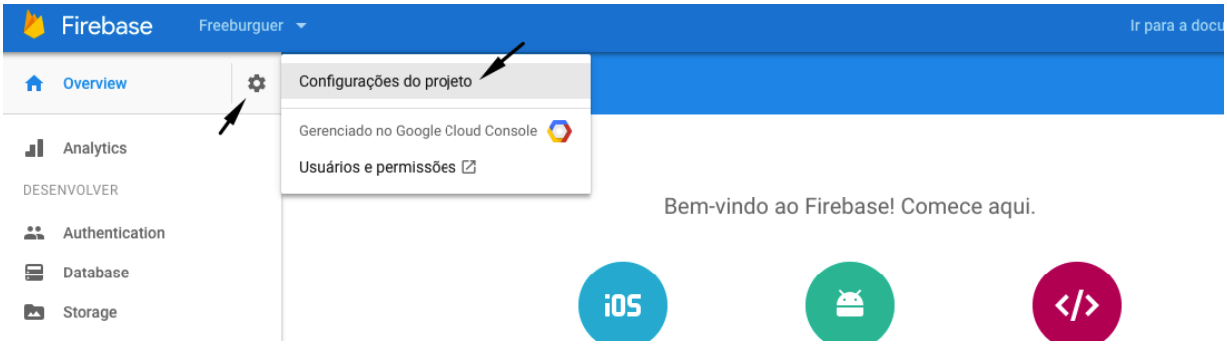


Figura 7.5: Site do Firebase

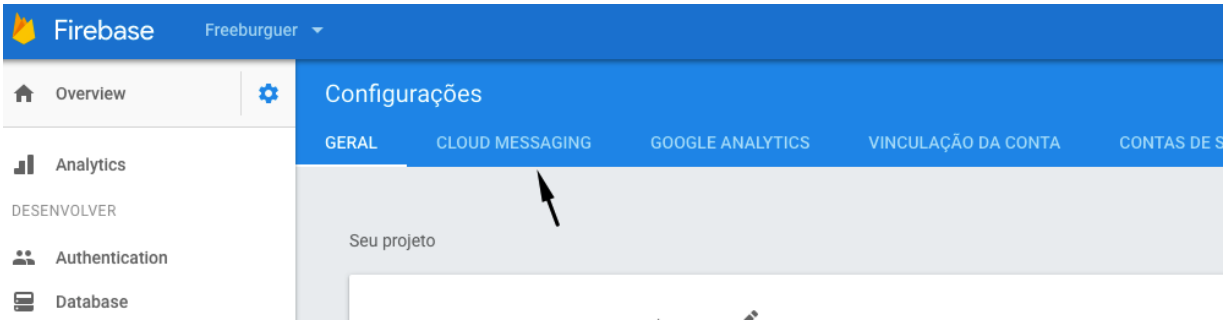
Selecione o projeto que deseja vincular as notificações:



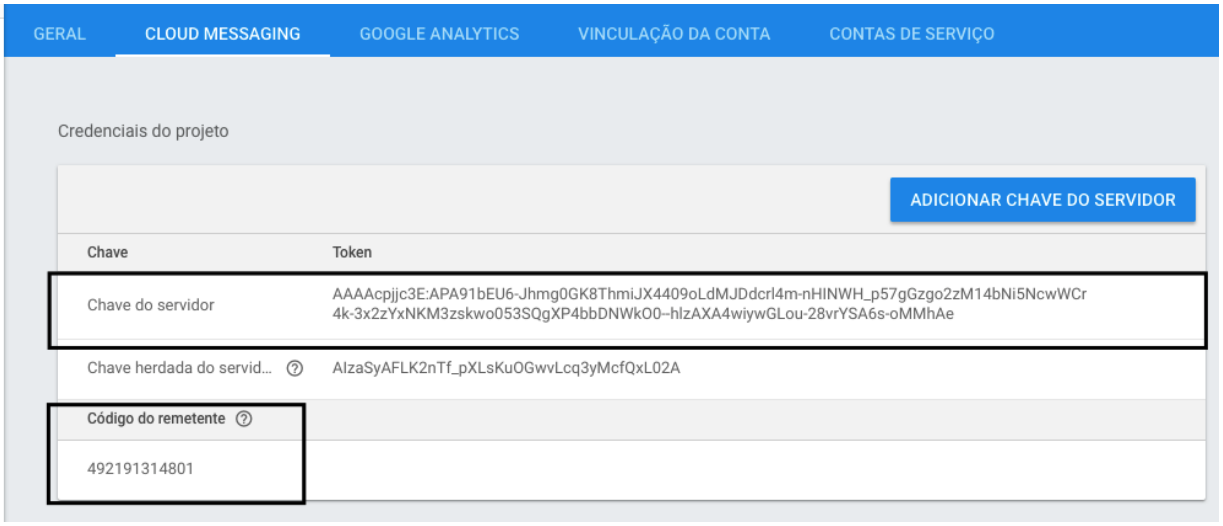
Navegue até `Configurações do projeto`:



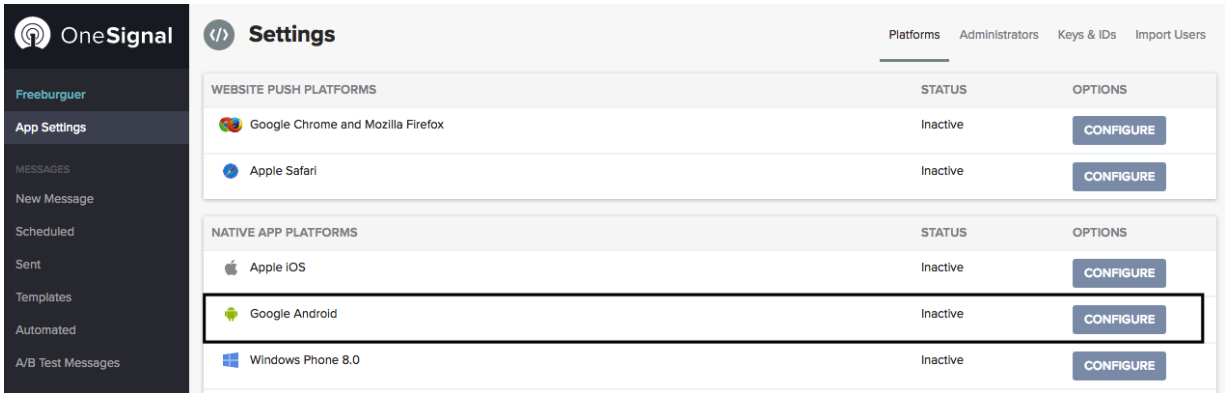
Em `Configurações`, clique em `Cloud Messaging`:



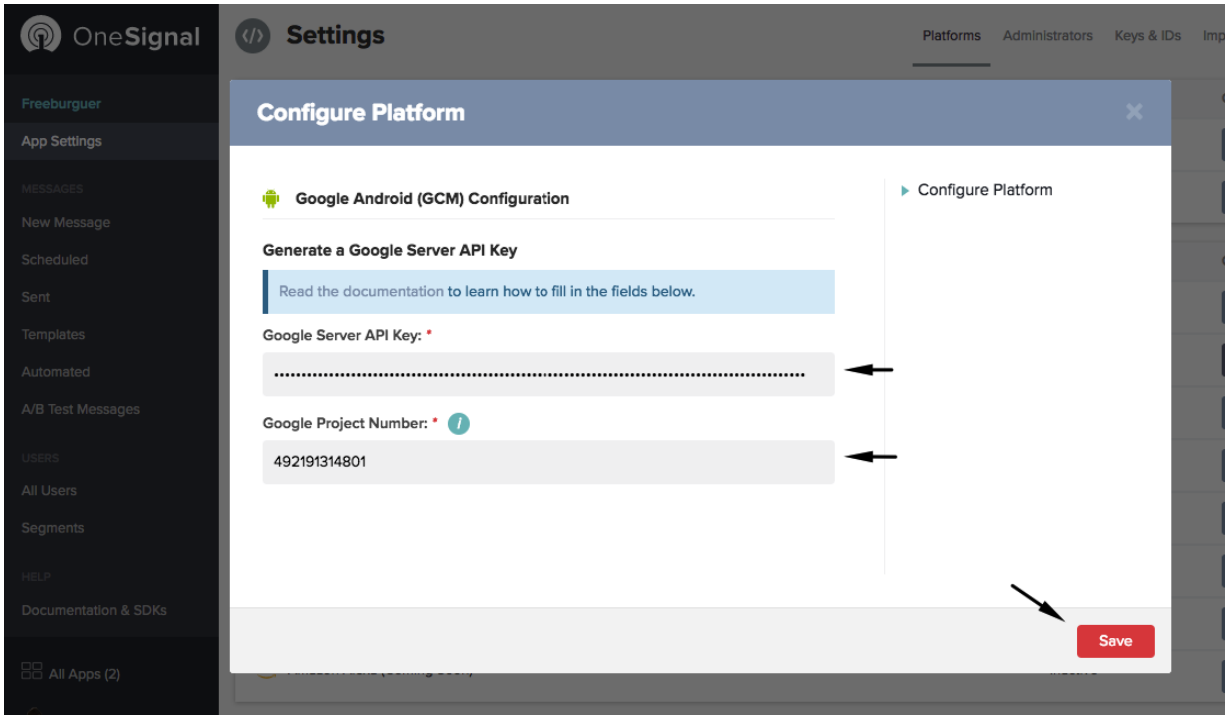
As informações sublinhadas na imagem a seguir são as que precisamos para colocar no site da OneSignal.



Acesse o site da OneSignal, faça seu cadastro e crie um projeto novo. Dentro do projeto, vá em `App Settings`, e selecione `Google Android`.



Lance as credenciais do projeto Android, para configurá-lo:



Com as credenciais configuradas, precisamos adicionar o plugin da OneSignal para Cordova ao projeto:

```
cordova plugin add onesignal-cordova-plugin --save
```

Vamos nos certificar agora de que os seguintes pacotes estão instalados no SDK do Android. Eles serão necessários para o funcionamento correto do Push Notification no Android:

- Android Support Repository
- Google Repository

Para isso, abra o *Android SDK Manager* com a linha de comando, ou usando o Android Studio, e certifique-se de ter esses pacotes instalados.

> **CUSTOMIZANDO O ÍCONE DA NOTIFICAÇÃO**
>
> É possível customizar o ícone que aparece no envio. Basta ler na documentação: https://documentation.onesignal.com/docs/customize-notification-icons.

Configurando na plataforma iOS

O primeiro passo para configurar a plataforma iOS é ativar o serviço de notificações dentro do App ID. Você deve ir até o https://developer.apple.com, na seção de `Certificates, Identifiers & Profiles` . Vá em `Identifiers -> App IDS` , selecione seu aplicativo (nosso caso, Freeburguer) e clique em `Edit` .

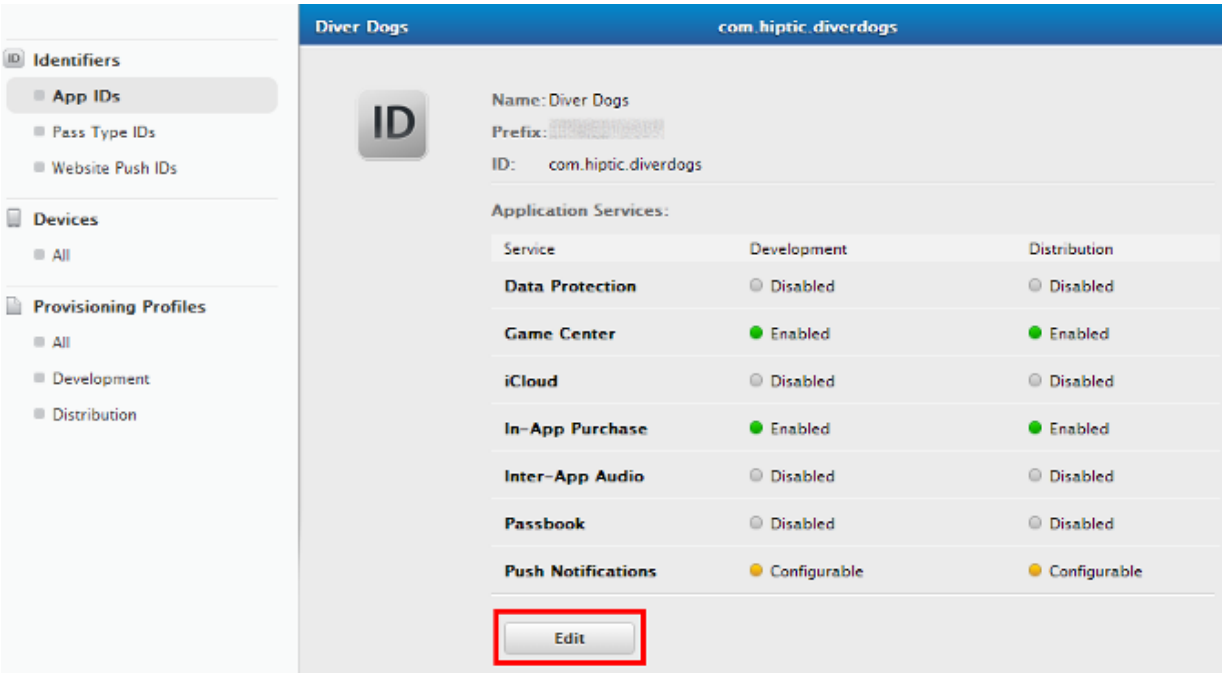


Figura 7.12: Imagem de exemplo

Encontre o serviço `Push Notifications`, ative-o e pressione `Done`. Agora vá até `Certificates -> All` e clique para criar um novo certificado. Escolha o tipo `Apple Push Notification Service SSL (Sandbox & Production)`.

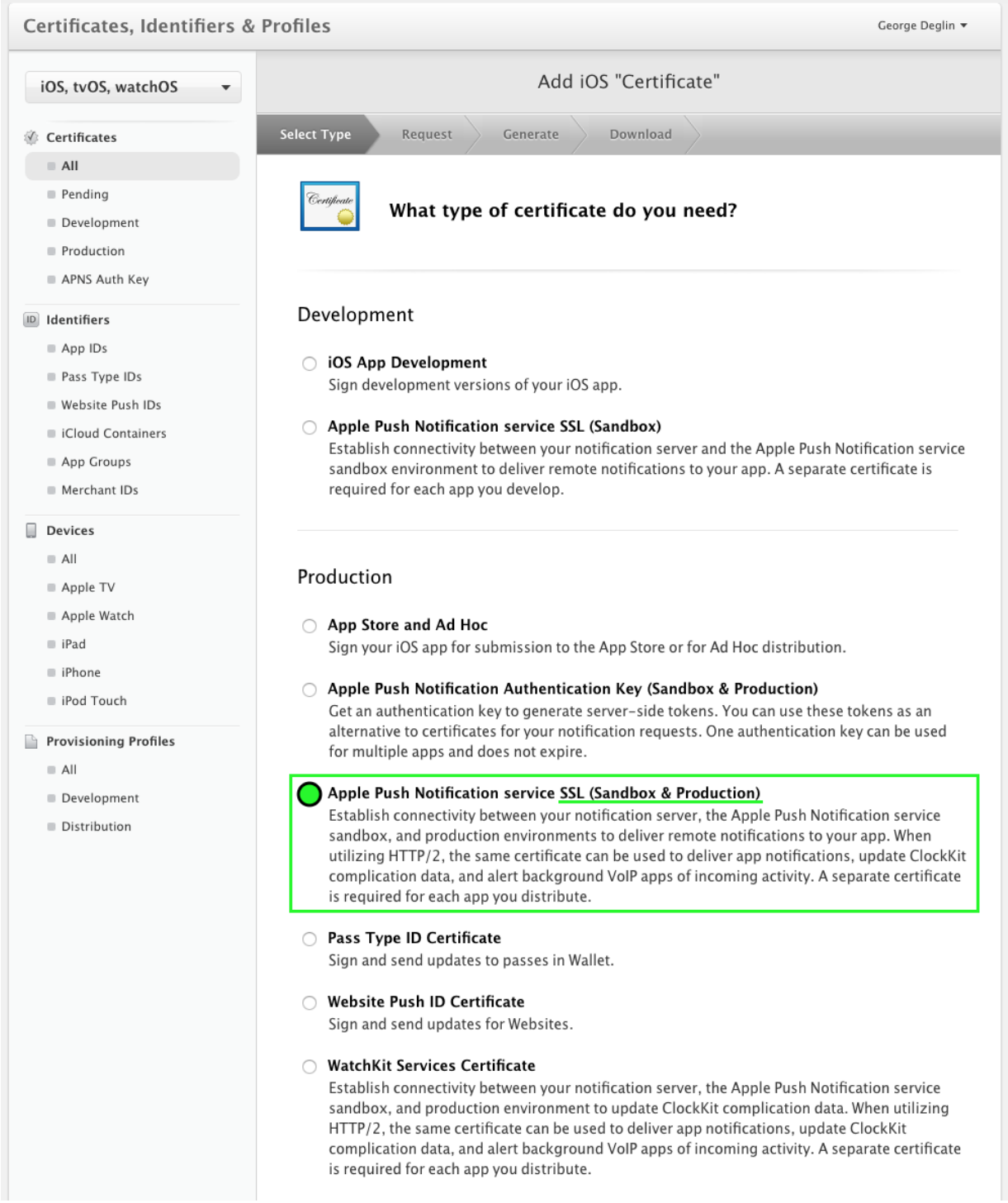

Certificates, Identifiers & Profiles
George Deglin
iOS, tvOS, watchOS
Certificates
All
Pending
Development
Production
APNS Auth Key
Identifiers
App IDs
Pass Type IDs
Website Push IDs
iCloud Containers
App Groups
Merchant IDs
Devices
All
Apple TV
Apple Watch
iPad
iPhone
iPod Touch
Provisioning Profiles
All
Development
Distribution
Add iOS "Certificate"
Select Type
Request
Generate
Download
What type of certificate do you need?
Development
iOS App Development
Sign development versions of your iOS app.
Apple Push Notification service SSL (Sandbox)
Establish connectivity between your notification server and the Apple Push Notification service sandbox environment to deliver remote notifications to your app. A separate certificate is required for each app you develop.
Production
App Store and Ad Hoc
Sign your iOS app for submission to the App Store or for Ad Hoc distribution.
Apple Push Notification Authentication Key (Sandbox & Production)
Get an authentication key to generate server-side tokens. You can use these tokens as an alternative to certificates for your notification requests. One authentication key can be used for multiple apps and does not expire.
Apple Push Notification service SSL (Sandbox & Production)
Establish connectivity between your notification server, the Apple Push Notification service sandbox, and production environments to deliver remote notifications to your app. When utilizing HTTP/2, the same certificate can be used to deliver app notifications, update ClockKit complication data, and alert background VoIP apps of incoming activity. A separate certificate is required for each app you distribute.
Pass Type ID Certificate
Sign and send updates to passes in Wallet.
Website Push ID Certificate
Sign and send updates for Websites.
WatchKit Services Certificate
Establish connectivity between your notification server, the Apple Push Notification service sandbox, and production environment to update ClockKit complication data. When utilizing HTTP/2, the same certificate can be used to deliver app notifications, update ClockKit complication data, and alert background VoIP apps of incoming activity. A separate certificate is required for each app you distribute.

Figura 7.13: Criando um certificado Push

Escolha o App ID do seu aplicativo no menu, e clique em `Continue`. O site vai lhe pedir agora sua identificação, criada já no capítulo anterior na seção *Criando um certificado*. Usaremos este arquivo na

seleção do `Upload CSR file`, então vamos selecionar `Choose File...` e clicar em `Generate`.

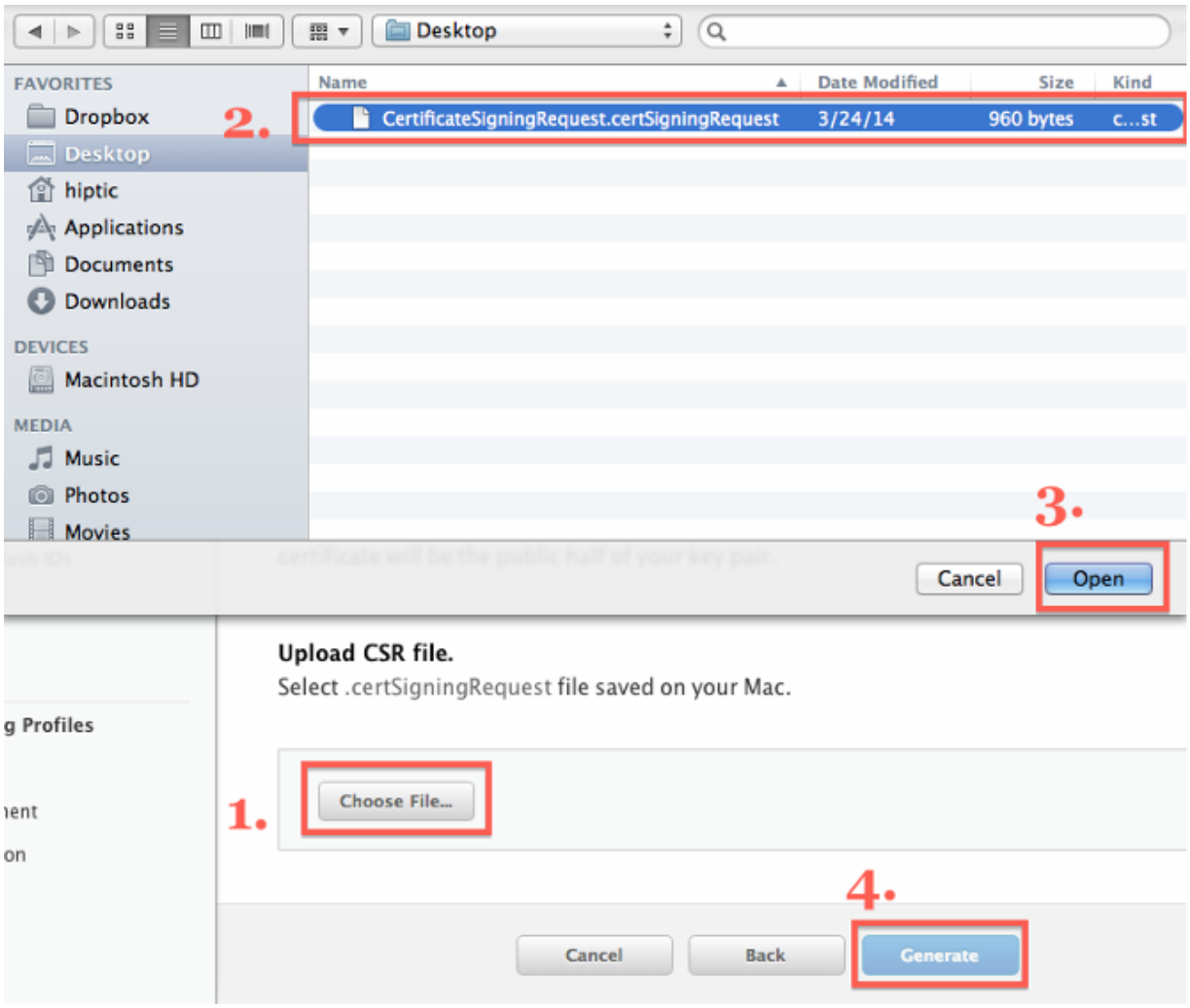
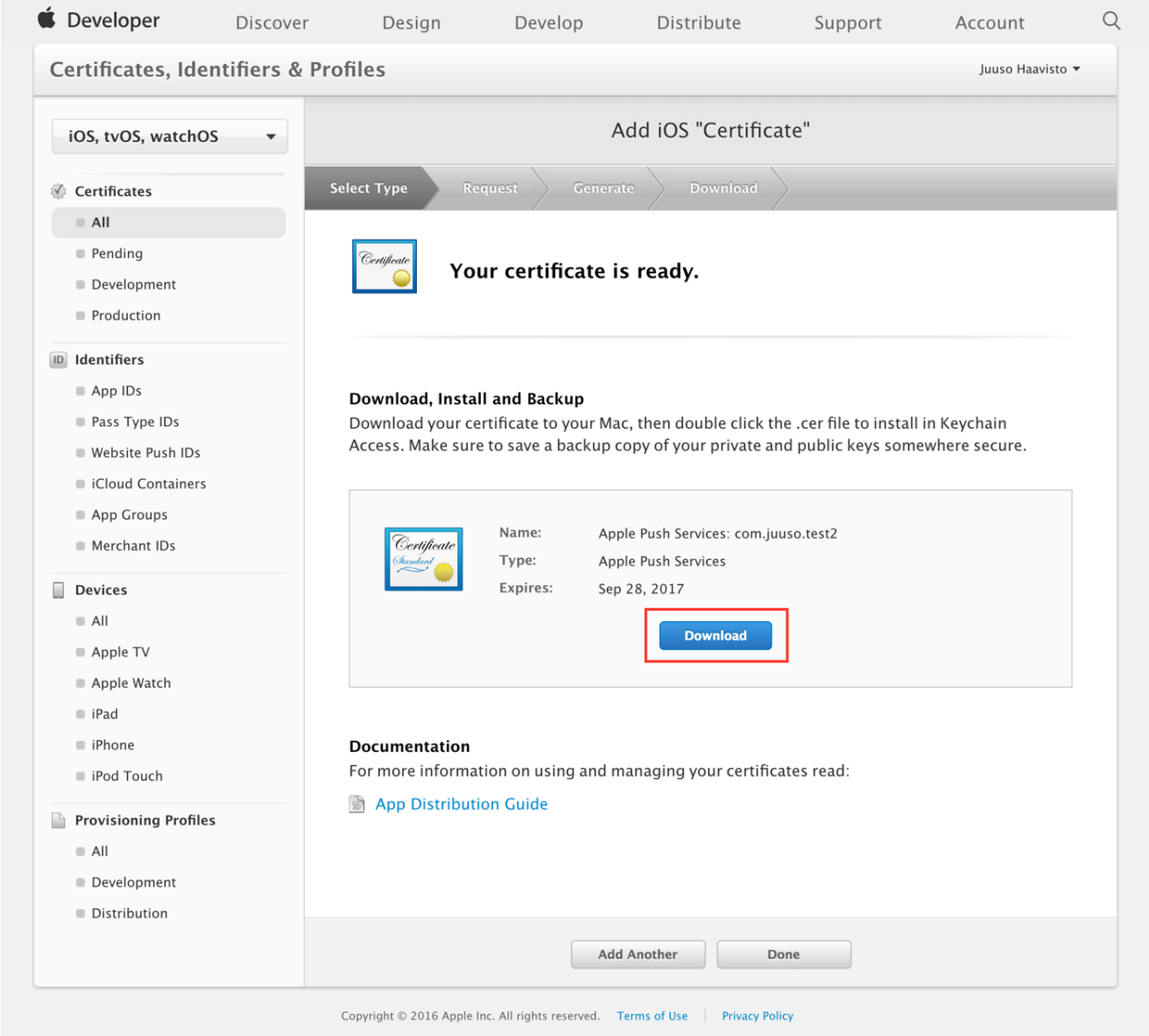
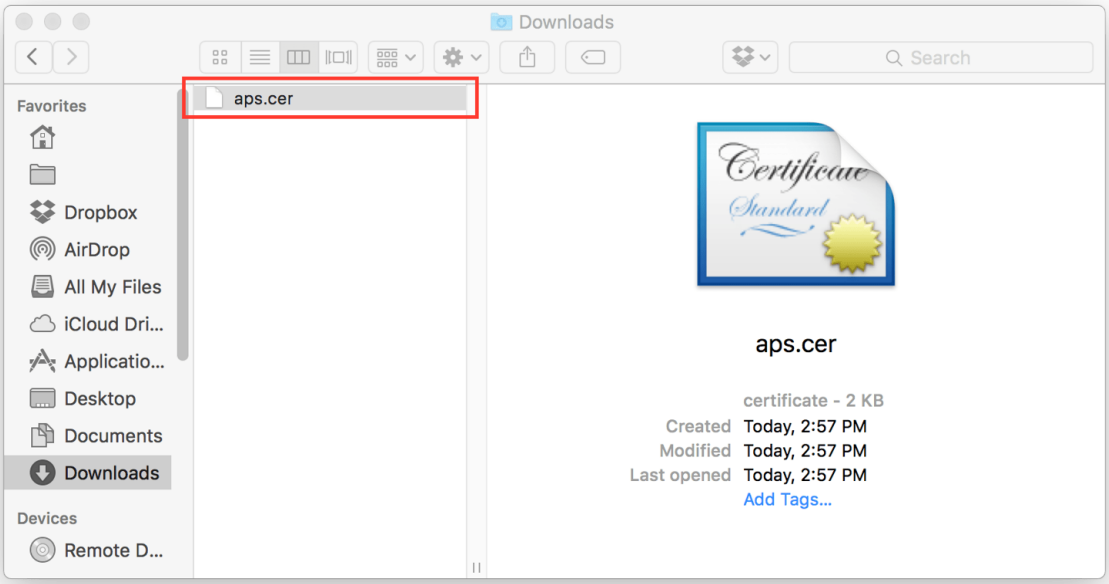


Figura 7.14: Selecionando o certificado de identificação

Faça o download do certificado gerado para a máquina.



Dê um duplo clique no certificado baixado para que ele abra no software `Acesso às chaves (Keychain Access)`.



No software, selecione seu certificado, clique com botão direito e pressione em `Export...` para exportar um arquivo com extensão `.p12`. Será esse arquivo que faremos upload dentro do OneSignal.

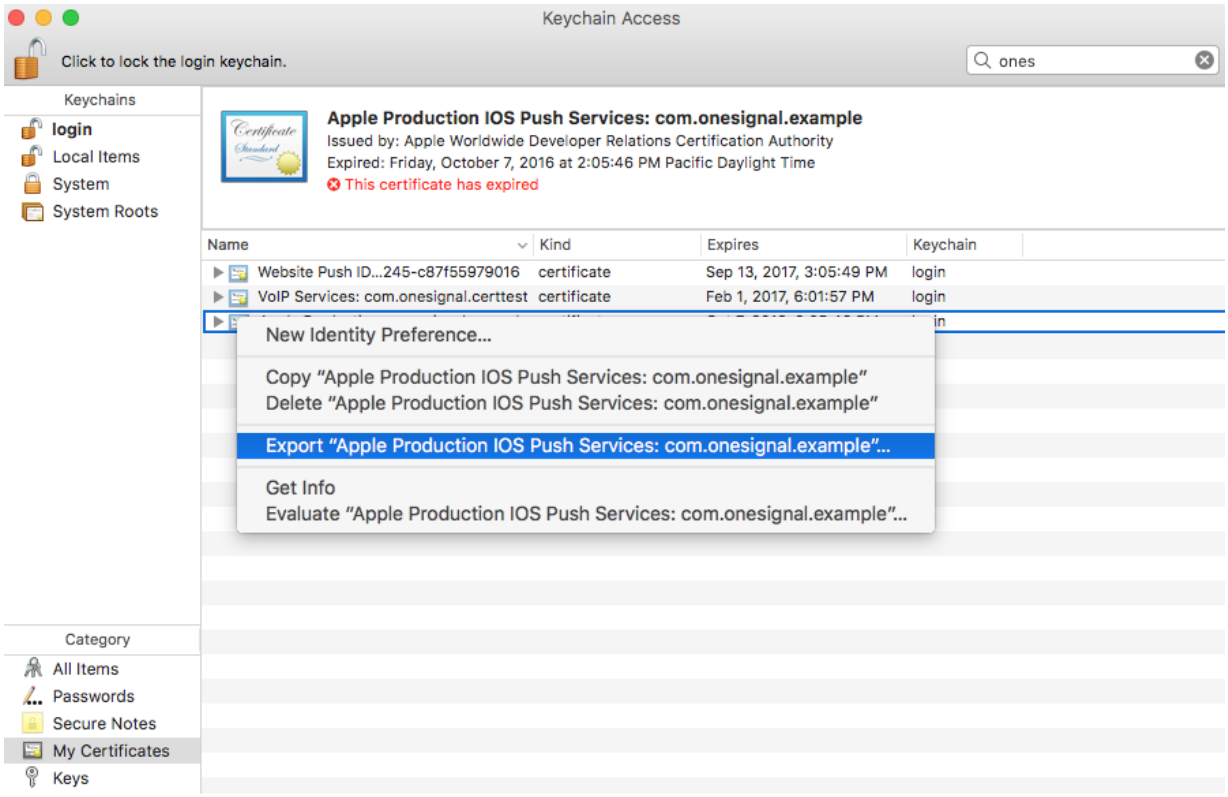


Figura 7.17: Etapa de exportar o certificado .p12

Vá novamente ao site da OneSignal, faça o login e selecione seu aplicativo. Dentro de `App Settings`, escolha `Apple iOS` e clique no botão `Configure`.

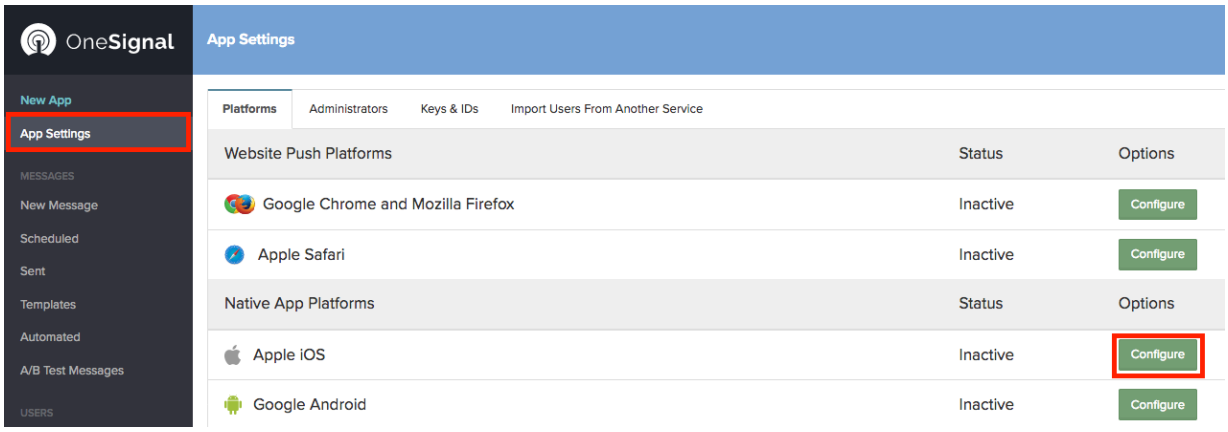
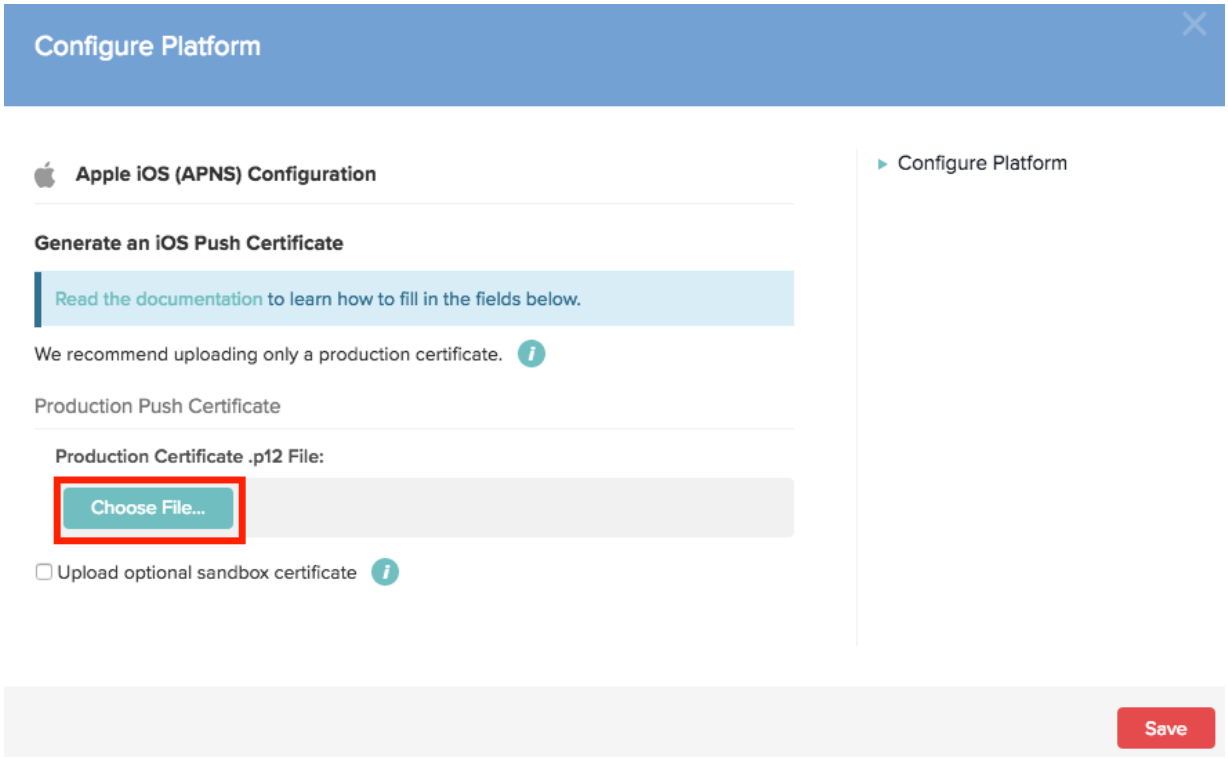


Figura 7.18: Configurando iOS na OneSignal

Será aberta uma janela, na qual você deve selecionar o arquivo exportado `.p12` para fazer upload. Pronto, o servidor da OneSignal já vai conseguir se comunicar com o da Apple, e estamos quase terminando.


Configure Platform
Apple iOS (APNS) Configuration
Generate an iOS Push Certificate
Read the documentation to learn how to fill in the fields below.
We recommend uploading only a production certificate.
Production Push Certificate
Production Certificate .p12 File:
Choose File...
Upload optional sandbox certificate
Configure Platform
Save

Precisamos nos certificar de ter o *CocoaPods* (gerenciador de dependências do Swift e Objective-C). No terminal, insira o comando a seguir:

```
sudo gem install cocoapods
pod repo update
```

Lembre-se de ter o plugin adicionado ao projeto:

```
cordova plugin add onesignal-cordova-plugin --save
```

Vamos agora ao Xcode. No diretório da plataforma iOS do seu aplicativo, depois de adicionar com `cordova plataforma add ios`, você terá um arquivo com o nome do seu aplicativo de extensão, como `.xcworkspace`. Clique duas vezes nele para abrir o projeto no Xcode.

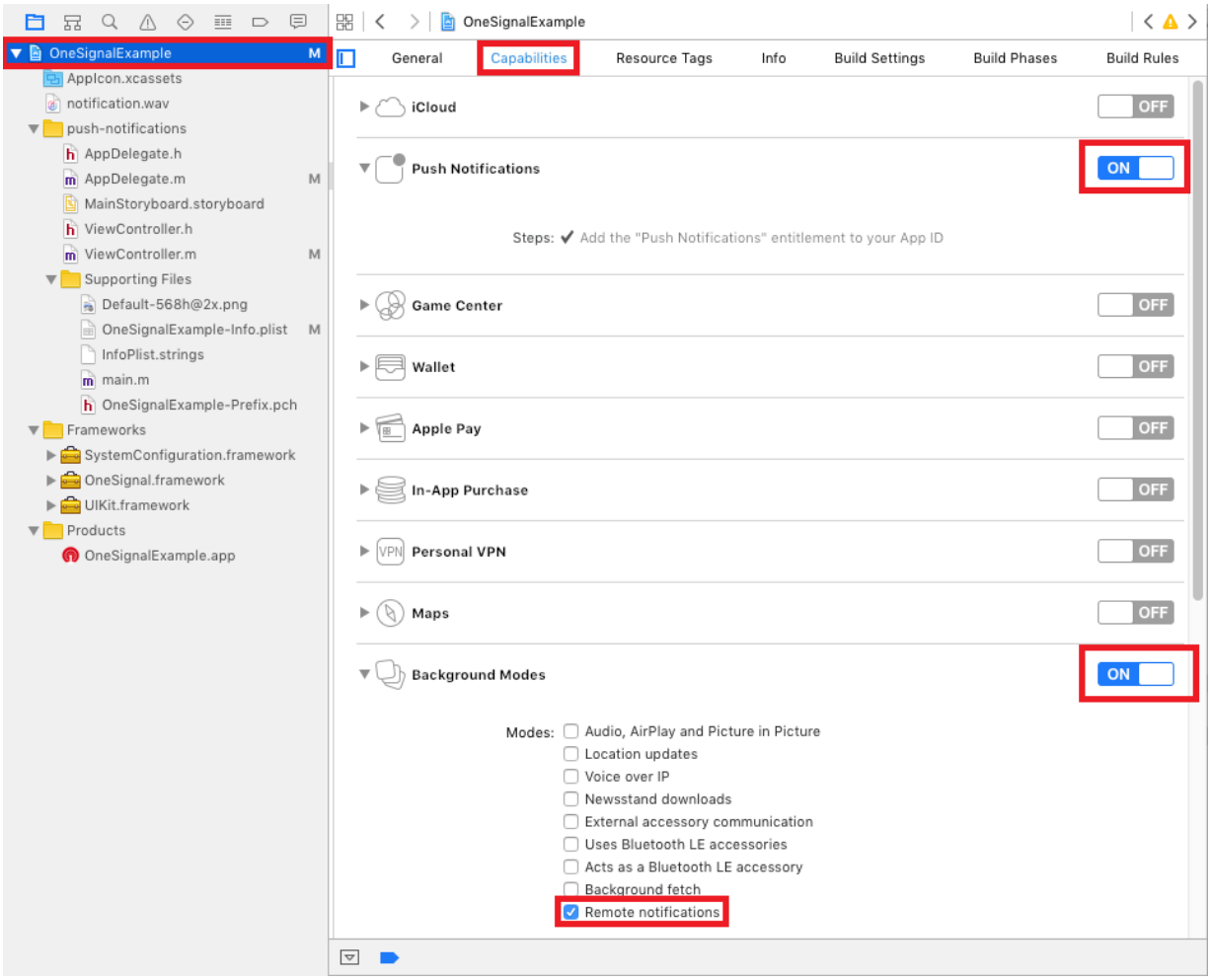


Figura 7.20: Locais para ativar dentro do projeto

Vá até `Capabilities`, e ative `Push Notifications` e `Backgrounds Modes -> Remote notifications`. Serviço configurado! Basta rodar a aplicação em um aparelho físico e enviar uma mensagem do painel da OneSignal para testar.

> Vale lembrar de que o simulador do Xcode não suporta notificações push, portanto, é necessário testar em um dispositivo real.

Registrando aparelhos ao iniciar

O último passo independente das plataformas é inscrever o aparelho no serviço OneSignal, dentro do código. Para isso, vamos usar do evento do Cordova `deviceready` como garantia de que o framework está carregado.

```
document.addEventListener("deviceready", function () {
}, false);
```

Vamos dar tempo para o AngularJS carregar. Por isso, encapsulamos a inicialização do plugin do OneSignal dentro de um `timeout()` de 2 segundos:

```
$timeout(function(){
    window.plugins.OneSignal
    .startInit("01fa3550-aa00-491d-a9bd-9e62a286501c")
    .endInit();
}, 2000);
```

O código dentro de `.startInit` é o **OneSignal App ID**. Você encontra-o dentro do painel do serviço, em `App Settings -> Keys & IDs`.

Enviando e recebendo notificações

Depois de abrir o aplicativo, você começará a visualizar os aparelhos registrados dentro do painel da OneSignal, em `Users -> All Users`. Um recurso bom para teste é adicionar os aparelhos que você usa para testar como `Test Users`.

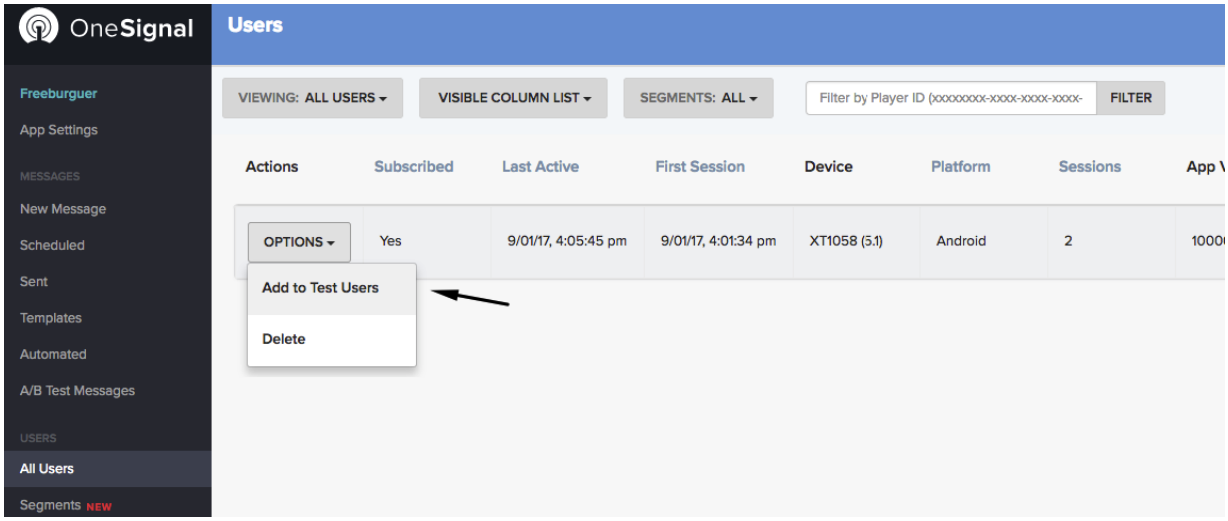


Figura 7.21: Adição de aparelho para grupo de teste

Vá até o painel da OneSignal, abra seu aplicativo e clique em `New Message -> Send to Test Device(s)`. Agora é só ir seguindo com `Next` para configurar as informações que lhe convém.

E se o aplicativo estiver aberto?

Se você receber uma notificação com o aplicativo rodando, você verá um `alert` nativo da plataforma.

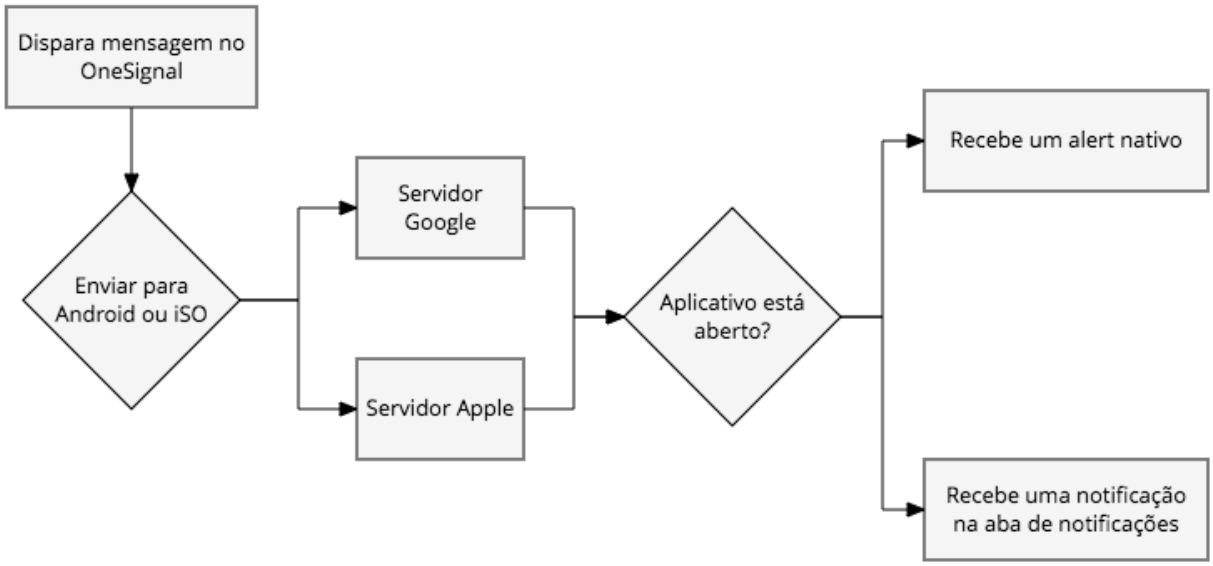


Figura 7.22: Fluxo de envio entre o OneSignal e o aparelho

Consumindo dados personalizados enviados do painel para os usuários

Um recurso bastante interessante é enviar dados customizados do painel para o seu aplicativo. Com isso, você pode enviar valores que serão consumidos pelo seu aplicativo assim que o usuário abrir a notificação. Assim, é possível customizar ações e funcionalidades dentro do app.

Para registrar um `callback` de quando o usuário abrir a mensagem, basta adicionarmos `handleNotificationOpened()` à inicialização, em que `abrirMensagem` é uma função que receberá os dados vindos do painel administrativo:

```
window.plugins.OneSignal
.startInit("01fa3550-aa00-491d-a9bd-9e62a286501c")
.handleNotificationOpened(abrirMensagem)
.endInit();
```

E a função `abrirMensagem` ficará desta forma:

```
var abrirMensagem = function(jsonRetorno) {
    console.log(jsonRetorno);
```

```
};
```

Legal, né? Uma das utilidades que já tive com isso foi redirecionar um usuário para uma determinada página do aplicativo, de acordo com o valor passado via Push Notification.

7.6 SVG: acrescentando vida à aplicação

O grande segredo dos componentes ricos na web hoje é o SVG. Por ser vetores, não perdem qualidade, possuem uma API de manipulação em JavaScript, são consideravelmente leves (pois carregam junto ao HTML), e estão presentes em muitas bibliotecas importantes na web.

> **SVG**
>
> É uma imagem vetorial no formato XML que suporta interatividade e animações.

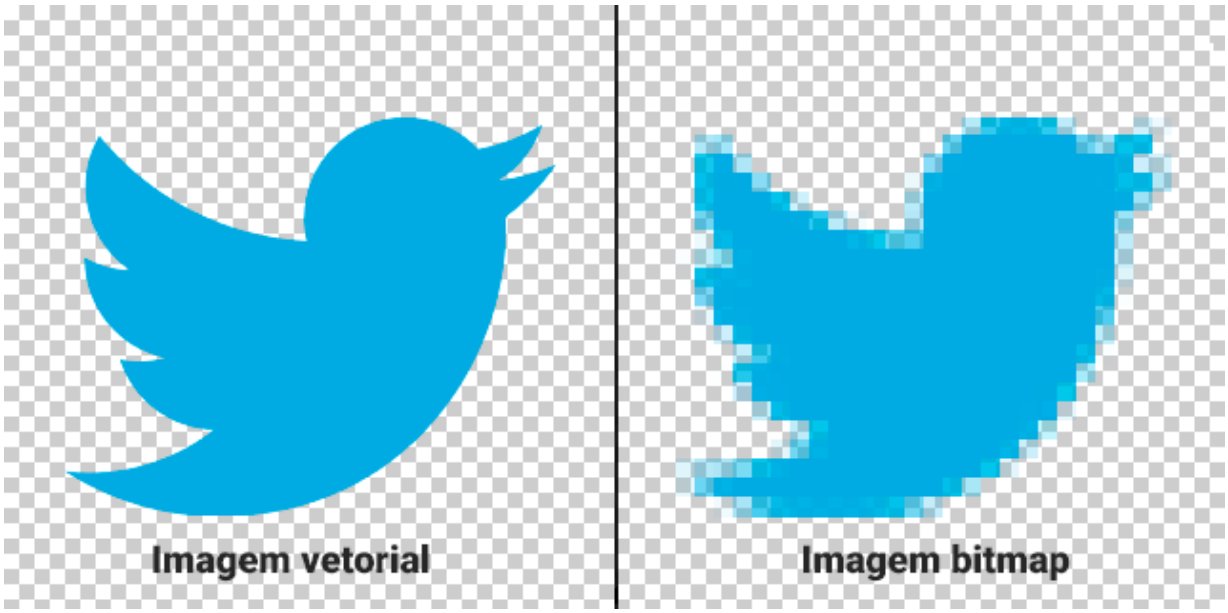


Figura 7.23: Diferença na qualidade de uma imagem vetorial e uma baseada em pixels

Entender ao menos um pouco sobre SVG pode ser um grande diferencial para você construir um aplicativo híbrido rico. Isso porque são muitas possibilidades de enriquecer a apresentação dos componentes do app. O objetivo aqui é abrir as possibilidades que o SVG traz dentro de aplicações híbridas, então, faremos duas animações de exemplo.

Formas básicas

Uma imagem em SVG é formada por formas básicas, como retângulos (`<rect/>`), círculos (`<circle />`), elipses (`<ellipse />`), linhas (`<line />`), conjunto de linhas (`<polyline />`) e polígonos (`<polygon />`).

Além dessas formas básicas, temos também a forma `<path/>` , que representa uma forma com curvas, linhas etc. Essa forma é a mais encontrada dentro de imagens SVG que você baixar ou criar em um software, como o Adobe Illustrator.

Se você inspecionar um SVG, você consegue perceber que é possível manipular cada elemento gráfico que forma a imagem. **Essa é a chave** para criarmos principalmente animações usando CSS.

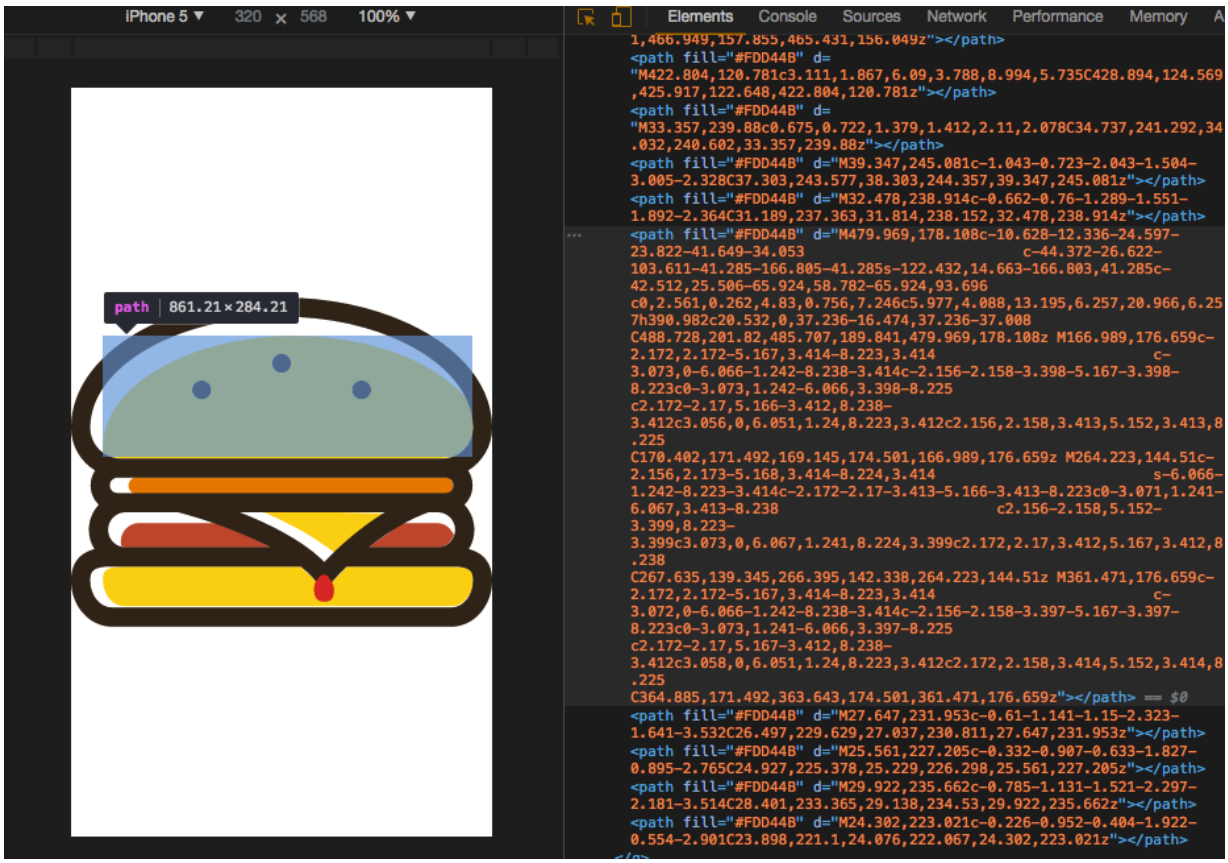


Figura 7.24: SVG inspecionado no Google Chrome

Animação: o pingo do ketchup

A ideia é fazer com que pingue uma gota de *ketchup* ao tocar o *input* de código da empresa. Assim, veremos uma gota escorrendo pela tela.

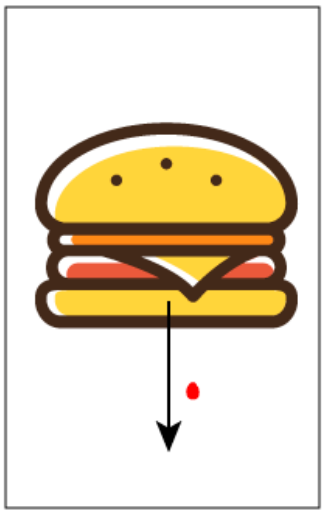


Figura 7.25: Ideia da animação

Para isso, a primeira coisa que fiz foi abrir o SVG no Illustrator e adicionar o formato da gota. Também mudei o nome do componente para `ketchup`, pois, quando exportar a imagem novamente, já terei como marcação esse ID.

Figura 7.26: SVG aberto no Illustrator

Depois que você cria uma imagem e manda salvar em SVG no Illustrator, uma janela vai se abrir como a figura a seguir.

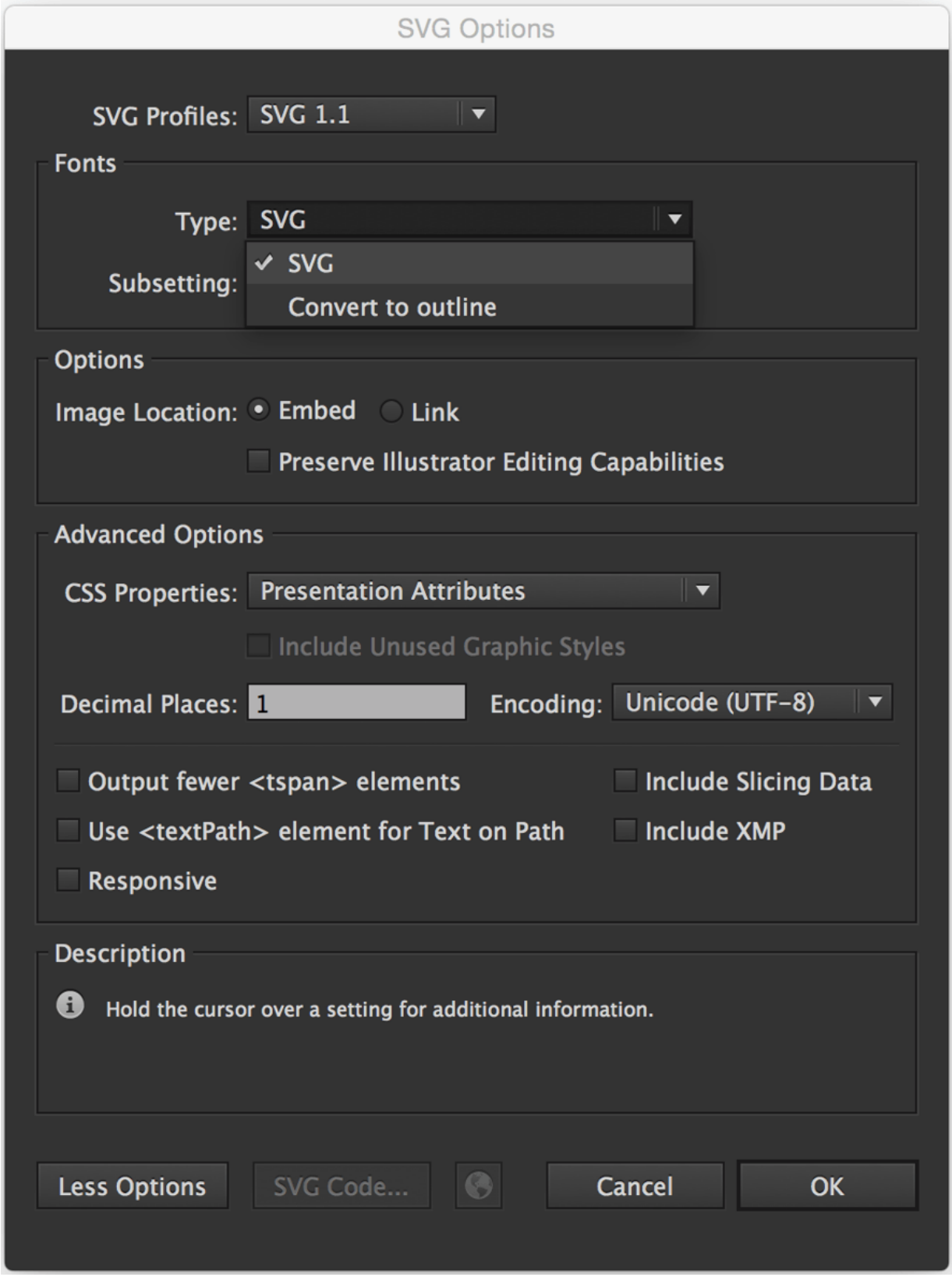

SVG Options
SVG Profiles:
SVG 1.1
Fonts
Type:
SVG
Subsetting:
SVG
Convert to outline
Options
Image Location:
Embed
Link
Preserve Illustrator Editing Capabilities
Advanced Options
CSS Properties:
Presentation Attributes
Include Unused Graphic Styles
Decimal Places:
1
Encoding:
Unicode (UTF-8)
Output fewer <tspan> elements
Include Slicing Data
Use <textPath> element for Text on Path
Include XMP
Responsive
Description
Hold the cursor over a setting for additional information.
Less Options
SVG Code...
Cancel
OK

Figura 7.27: Configurações mais recomendadas

A opção de `CSS Properties` é importante, porque, caso você escolha para que fique em uma tag `<style>`, podemos correr o risco de outros SVGs possuírem uma mesma classe e acabarmos quebrando outros elementos. Então, defina sempre os elementos com prefixos específicos. Se você escolher *presentation attributes*, tudo já estará no elemento e facilitará sua vida.

Para o valor `Decimal Places`, quanto menor o número de casas decimais, menor será o tamanho do SVG. Em geral, uma casa decimal já é o suficiente.

Com a imagem exportada, podemos abrir literalmente o SVG no editor de código. Você verá todo o XML que forma a imagem. Basta copiar e colar dentro da área do HTML que você deseja que apareça – no nosso caso, na View `home.html`.

A ideia agora é esconder o *path* com ID `#ketchup` até que o usuário pressione-o. Para isso, usamos um simples `display: none;`:

```
#ketchup{
    display: none;
}
```

Agora, vamos criar uma classe CSS que exibe esse *path* e faz a animação dele "*caindo*" na tela:

```
#ketchup.pingar{
    display: block;
    animation: pingar 2s cubic-bezier(0.645, 0.045, 0.355, 1.000) both;
}
```

O próximo passo é a animação `pingar`, da seguinte maneira:

```
@keyframes pingar{
    55%{
        transform: translateY(10px);

    }
```

```
    100%{
        transform: translateY(645px);
    }
}
```

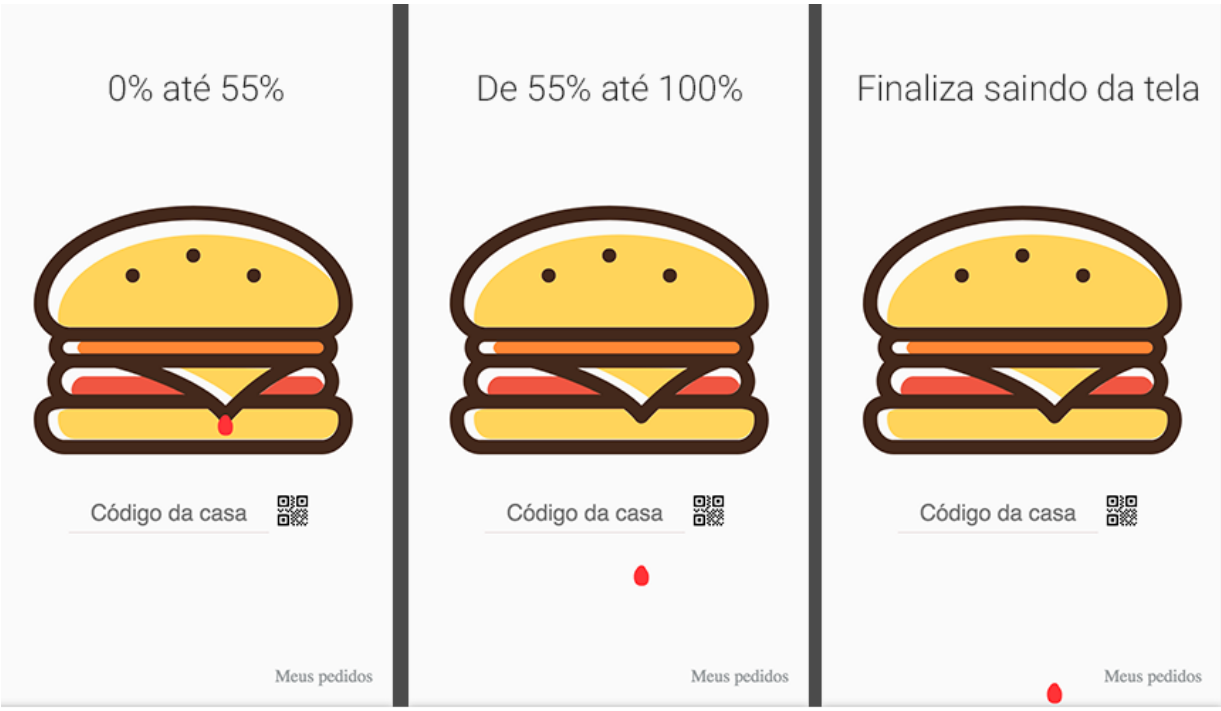


Figura 7.28: Transição em execução

Com ajuda do AngularJS, na View `home.html`, adicionamos a chamada para uma função `ng-click="Home.pingar()"` ao input `codigo_empresa`. Essa função adiciona a classe `.pingar`, que faz a animação CSS.

Dentro do controlador `HomeController`, declaramos a função `pingar()`:

```
this.pingar = function(){
    var element = document.querySelector('#ketchup');
    element.setAttribute("class", "pingar");
}
```

Pronto! De forma simples, fizemos uma animação. É claro que temos como otimizar o SVG e melhorar a ideia da animação, mas,

de modo geral, é importante que você perceba o que o SVG pode fazer pela interação da sua aplicação.

Animação: veículo do pedido entrando

Outra animação boba que vamos fazer é a de um veículo entrando na tela e parando de forma brusca.

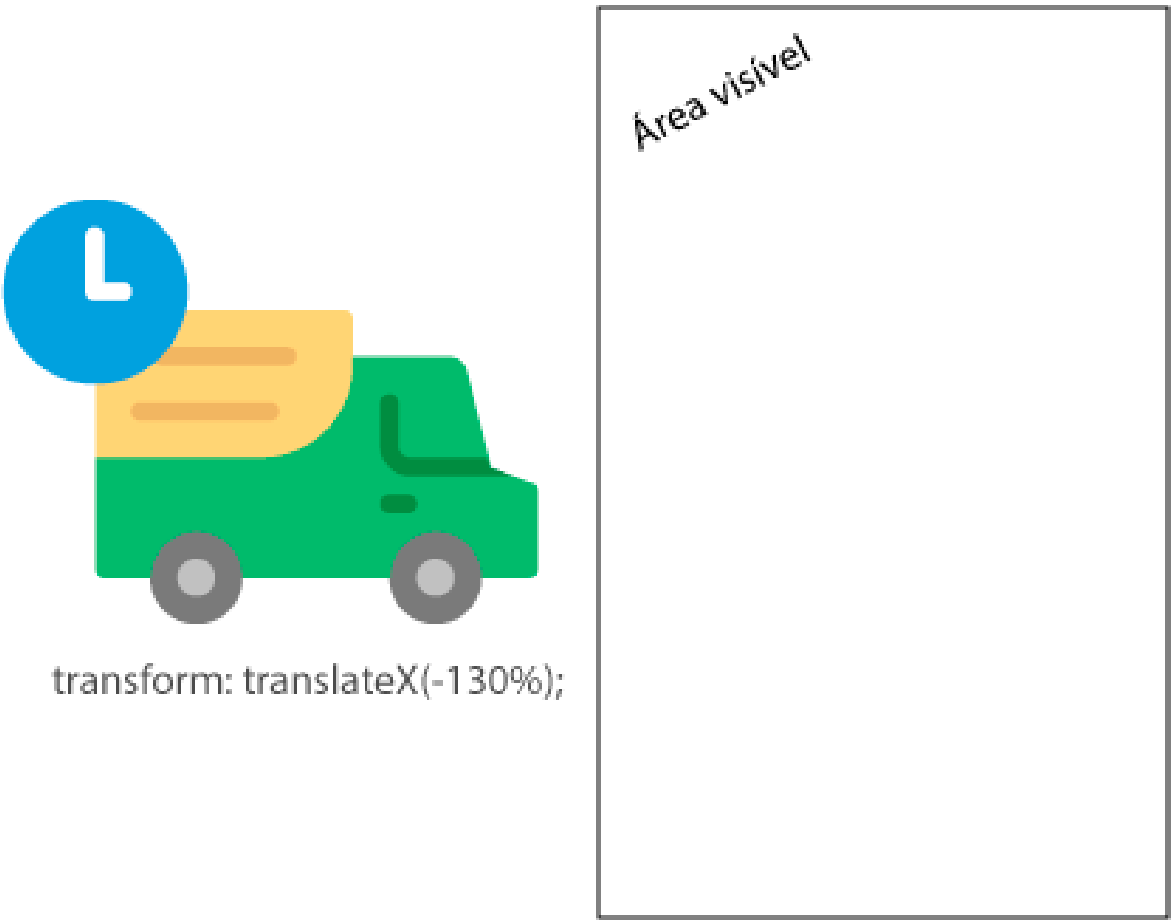


Figura 7.29: Ideia para animação

O SVG do veículo estará **escondido** da visualização usando um `translateX(-130%)` e `opacity: 0` ; e a animação fará o papel de trazer de volta à tela. A primeira coisa que fiz foi definir um ID para a marcação SVG na View `pedido-busca.html` . O SVG tem um ID `#svg-truck` , e nele definimos as propriedades a seguir:

```
#svg-truck{
    opacity: 0;
    transform: translateX(-130%);
    animation: entrar 1s cubic-bezier(0.645, 0.045, 0.355, 1.000) both;
    animation-delay: 200ms;
}
```

Tempo da animação cubic-bezier

No código anterior, usamos uma função `cubic-bezier`, uma das funções de curva de aceleração que o CSS aceita. Para entender melhor, temos a animação percorrendo de forma linear o tempo e, com as funções de aceleração, podemos customizar o desenrolar do tempo para definir como o elemento vai progredir na animação dentro de um segundo. O site http://bit.ly/2GotvWP é uma ferramenta muito boa para você testar e copiar o código CSS pronto de diferentes configurações de tempo.

Repare que esperamos 200 milissegundos para executar a animação com um `animation-delay`. Isso é legal porque dá tempo da DOM do navegador carregar e não travar a tela. Também notamos no código anterior que temos a chamada da `animation: entrar`, e a animação foi escrita da seguinte forma:

```
@keyframes entrar{
    85%{
        transform: rotate(-15deg);
    }
    100%{
        opacity: 1;
        transform: translateX(0%);
        transform: rotate(0deg);
    }
}
```

O interessante é que, nos 85% da animação, fizemos uma pequena inclinação para dar uma impressão de uma alta aceleração e de

uma freada brusca. Pronto, o objetivo foi concluído com poucas linhas e uma experiência mais rica.

Indo além com SVG

SVG é um assunto que vale a pena ser estudado mais a fundo. Há muito a se aprender e, quanto mais você souber, mais possibilidades você tem de criar aplicativos incríveis.

Usamos bastante o `npm` neste livro, existe uma ferramenta que você pode utilizar para otimizar o tamanho das suas imagens SVG baixadas, o `svgo` . Para instalar, basta rodar no terminal:

```
npm install -g svgo
```

Seu uso é bastante simples. Com o terminal aberto, você roda o comando:

```
svgo estrela.svg -o estrela-otimizada.svg
```

O código anterior vai otimizar e tirar tudo o que for desnecessário da sua imagem SVG. Essa ferramenta é bastante útil, pois qualquer economia é válida quando estamos falando de otimização.

Para aprender ainda mais, existe uma lista criada pelo desenvolvedor *William Justen* que reúne uma quantidade sensacional de conteúdos relacionados ao assunto. É só acessar: https://github.com/willianjusten/awesome-svg.

> Você pode baixar o aplicativo pronto até aqui no link http://bit.ly/cordova-livro). O projeto é o **freeburguer-cap7**.

7.7 Ative o modo produção do AngularJS

Como usamos o framework SPA para nos ajudar no aplicativo, uma última dica para otimizar a aplicação é desativar o modo `Debug Data` do AngularJS. Para fazer isso, vamos até o arquivo `bootstrap.js` e, dentro da declaração `.config()`, injetamos o `$compileProvider`.

Então, dentro da função, desativamos o modo debug. Isso desligará uma série de verificações e códigos extras que o framework usa por trás da aplicação – úteis em desenvolvimento, mas não em produção.

```
.config(function($routeProvider, $compileProvider){
    $compileProvider.debugInfoEnabled(false);
}
```

7.8 Revisão

Neste capítulo, aprendemos uma série de assuntos relacionados ao desenvolvimento híbrido com Cordova, que juntos oferecem melhorias significativas para o resultado final do seu aplicativo. Vimos como melhorar a experiência do usuário com animações SVG, e também automatizamos uma tarefa específica com os hooks do Cordova. Aprendemos como configurar as famosas notificações *push* e conhecemos os parâmetros essenciais para garantir segurança dentro de um aplicativo Cordova.

Capítulo 8
Continuando os estudos

Chegamos ao fim da jornada. Espero que o conhecimento deste livro possa influenciar sua carreira e no modo como você trabalha, de forma positiva. Há muito para estudar e em quese aprimorar. Estamos vivendo uma época incrível para o desenvolvimento híbrido. Com aparelhos para a WebView sempre atualizados, podemos esperar do futuro um suporte às melhores funcionalidades web que estão surgindo.

Também temos hoje um conjunto sólido de boas ferramentas para construir aplicações SPA (*Single Page Application*), como React, Vue.js, AngularJS e o novo Angular, entre outros. Além disso, temos frameworks como Ionic que oferecem ferramentas incríveis para o desenvolvimento de aplicativos. Mas lembre-se sempre de ter uma base sólida no conhecimento-chave: **HTML**, **CSS** e **JavaScript**.

8.1 O que esperar do PhoneGap para o futuro

Em 2017, a equipe do PhoneGap fez um post explicando como seria o desenvolvimento da ferramenta deste ano em diante. Algumas mudanças significativas foram apresentadas e, entre elas, estão alguns pontos fortes:

- Implementar mais projetos de inicialização usando os frameworks mais conhecidos atualmente, para facilitar o início rápido.
- Cordova e PhoneGap serão cada vez mais definidos. Começamos o livro explicando que os dois são uma coisa só, mas o fato é que, agora, a equipe do PhoneGap quer separar as coisas mais do que nunca, e vai continuar usando e dando

suporte ao código-fonte do Apache Cordova para ser seu núcleo de acesso a APIs nativas.
- Progressive Web Apps (PWA): a ideia é que os mesmos aplicativos móveis que você lança hoje sejam portados para web no futuro usando PWA.

Em resumo, o PhoneGap vai focar na entrega de ferramentas que ajudam ainda mais no desenvolvimento híbrido, e também pensa em melhorar a facilidade em iniciar projetos com os diversos frameworks JavaScript do mercado (React, Vue.js e Angular). Por fim, teremos o suporte de portar um aplicativo para PWA.

O projeto Apache Cordova continua como núcleo para vários outros projetos; para o PhoneGap, também não muda, é ele quem adiciona as APIs para acesso nativo nas plataformas.

8.2 Como resolver problemas de dependências e erros

Às vezes, nem tudo roda como queremos. Situações inesperadas podem acontecer ao tentarmos instalar um pacote no NPM, atualizar um plugin, fazer o `build` da aplicação, entre outros problemas comuns. Os problemas inevitavelmente vão aparecer, mas você deve saber onde e como buscar ajuda.

> **Aproveite o fórum da Casa do Código**
>
> A Casa do Código possui um fórum exclusivo para você, leitor, poder compartilhar experiências e tirar dúvidas: http://forum.casadocodigo.com.br.

Outras opções que você tem para estudo são:

- **Repositório oficial** – Olhar o repositório oficial do Cordova ou do Phonegap na seção *Issues* pode ajudar, já que a comunidade de software livre é bastante ativa:
  - https://github.com/phonegap
  - https://github.com/apache
- **StackOverflow** – Um bom lugar para se buscar ajuda e ver resoluções de problemas, e que possui comunidade bastante ativa: http://stackoverflow.com/questions/tagged/cordova
- **Redes sociais** – Outra opção são as comunidades ou grupos das redes sociais. Existem desenvolvedores bem ativos e dispostos a ajudar lá:
  - Comunidade de PhoneGap no Facebook – https://www.facebook.com/groups/PhoneGapdev
  - Comunidade brasileira de PhoneGap no Facebook – https://www.facebook.com/groups/PhoneGapbr
- Por último, também existem os grupos:
  - Grupo no Google Groups – https://groups.google.com/forum/#!forum/phonegap
  - Comunidade Adobe – https://forums.adobe.com/community/phonegap

8.3 Links úteis

Separei nesta seção uma série de links que podem ser úteis para você continuar seus estudos.

Recursos web úteis

A seguir, deixo alguns links de ferramentas que utilizo quando estou criando um novo aplicativo:

- Ícones – https://www.flaticon.com
- Ícones material design – https://material.io/icons
- Animações CSS3 prontas – http://animista.net/play/basic

- Sistema de Grid completo com FlexBox parecido com Bootstrap – http://flexboxgrid.com
- Ferramenta para encontrar tons de uma mesma cor hexadecimal – http://www.0to255.com
- iTunes Link Maker, que ajuda a gerar o ícone de download do seu app iOS – http://apple.co/2yuNvXu
- Editor JSON online – http://www.jsoneditoronline.org
- Gerador de breakpoints responsivos – http://simplecss.eu/index.html
- Plataforma Firebase – https://firebase.google.com

Blogs e cursos

- Blog cheio de conteúdo exclusivo, da *Holly Schinsky*, desenvolvedora da Adobe – http://devgirl.org
- Blog com tutoriais – http://phonegap-tips.com
- Curso gratuito de Phonegap – https://loiane.training/course/phonegap-apache-cordova
- Curso da plataforma Alura – https://www.alura.com.br/curso-online-cordova-phonegap

Obrigado por ler este livro. Espero que você construa aplicações incríveis!

Capítulo 9
Apêndice A – Configurando seu ambiente

Trabalhar com desenvolvimento híbrido envolve testar o seu aplicativo em diversas plataformas. Se dependêssemos apenas dos dispositivos físicos para testes, o custo seria altíssimo – embora seja recomendado ter um aparelho para teste real em cada plataforma. Com as mudanças e os lançamentos de novos dispositivos, fica difícil acompanhar, logo, você vai precisar dos emuladores.

Mais à frente, veremos que, apesar de podermos usar alguns serviços como o *PhoneGap Build* e *PhoneGap App* para testar nossas aplicações e ganhar mais produtividade, eles ainda deixam a desejar em muitos quesitos. **Nada substituirá o teste real da aplicação** no aparelho ou nos emuladores.

Sérgio Lopes, em seu livro *Aplicações mobile híbridas com Cordova e PhoneGap* (2016), fala que essas ferramentas *"... até tentam e suportam muitos plugins oficiais do Cordova e PhoneGap, mas você provavelmente vai acabar usando algum plugin externo alguma hora*".

Assim, o objetivo deste apêndice é deixar a sua máquina pronta para *emular* com emulador e fazer a *compilação* correta da aplicação para cada plataforma.

Instale o Git

O Git é um sistema de controle de versão fortemente utilizado no mercado. Apesar de ele ser considerado *opcional* pela documentação do Cordova, eu considero-o fundamental. O Git será usado pelo *CLI (Cliente de linha de comando)* do Cordova e do PhoneGap para baixar arquivos de repositórios, e aqui no livro será disponibilizado um link para download (ou *clone*) do projeto por ele.

Repositório do livro

Baixe os projetos de cada capítulo no repositório do GitHub, em: http://bit.ly/cordova-livro.

Como o Git tem a instalação diferenciada para cada sistema operacional, e não é nosso objetivo descrever essa instalação aqui, seguem alguns links para ajudá-lo nisso:

- Site oficial – https://git-scm.com
- Guia prático da instalação e comandos básicos – http://bit.ly/guia-git
- Minicurso interativo em inglês – http://try.github.io

Quer saber mais sobre Git?

A Casa do Código possui um livro chamado *Controlando versões com Git e GitHub*, dos autores Alexandre Aquiles e Rodrigo Ferreira (2014).

Fique atento à instalação do Node.js

O Node.js é um interpretador de código JavaScript que funciona no lado do servidor. Geralmente, ele vem com um gerenciador de pacotes chamado NPM (*Node Package Manager*), sendo que, por meio dele, instalamos os pacotes necessários do Cordova e do PhoneGap.

Também existem outros excelentes pacotes via NPM que podemos instalar para trabalhar em conjunto no desenvolvimento híbrido. Como exemplo temos o Gulp (automatizador de tarefas) e o Browser-sync (ajuda a sincronizar em tempo real uma página web em vários dispositivos).

É importante ficar atento a alguns detalhes:

- **Usuários Linux Ubuntu** – Nas distribuições Ubuntu, devido a um conflito com outro pacote do sistema, o comando no terminal chama-se `nodejs`, e não `node` como nos demais.
- **Versão LTS ou Current** – Na documentação oficial do Cordova, não cita qual versão devemos instalar. A versão *LTS* é aquela com suporte prolongado, possui um ciclo de atualizações diferente da versão *Current* e é direcionada para a maioria dos usuários. Já a *Current* (corrente, atual) é para quem deseja utilizar as últimas funcionalidades lançadas.

No livro, vamos utilizar a versão LTS, visando o suporte de todos os pacotes necessários para desenvolver com Cordova e PhoneGap. Se quiser entender melhor sobre o suporte LTS, acesse https://github.com/nodejs/LTS.

SDKs e as plataformas

Antes de adentrarmos as configurações, acho importante você ter uma visão rápida das plataformas quanto ao seu SDK, público e à sua flexibilidade:

- **Android**
  - SDK roda multiplataforma (Windows, macOS e Linux);
  - É a plataforma mais flexível para se desenvolver;
  - No Brasil, o Android é o sistema operacional mais popular.
- **iOS**
  - SDK roda somente no macOS;
  - É a plataforma menos flexível para se desenvolver, porque depende de você ter um Macbook;
  - No Brasil, o iOS é o sistema mais popular depois do Android. Não é regra, mas geralmente atinge um mercado com nível aquisitivo maior pelo custo do iPhone ser elevado.

Essa questão do SDK do Windows não rodar no Linux, por exemplo, não nos impede de programar a aplicação utilizando uma máquina Linux. Porém, ao final, você precisará executar um comando em cima da plataforma que vai gerar o seu aplicativo. Neste momento, entram essas restrições nos casos do iOS e do Windows Phone, como só podermos rodar o seguinte comando em uma máquina que roda macOS: `cordova platform add ios .`

A seguir, vamos falar de configurações específicas. Sinta-se à vontade para pular para aquela que se encaixa na sua realidade, **plataforma** e **sistema operacional** que usa ou deseja configurar.

9.1 Android no Linux

O sistema operacional Android é desenvolvido em cima de uma API estruturada na linguagem Java, por isso, a documentação do Cordova recomenda instalar o Oracle Java JDK (*Java Development Kit*). Ele será utilizado pelo Android SDK (*kit de desenvolvimento de software*) a todo momento.

Verifique se sua máquina já possui o `javac` (compilador da linguagem Java), disponível quando já temos o JDK instalado. Para isso, abra o terminal do Linux e rode o comando:

```
javac -version
```

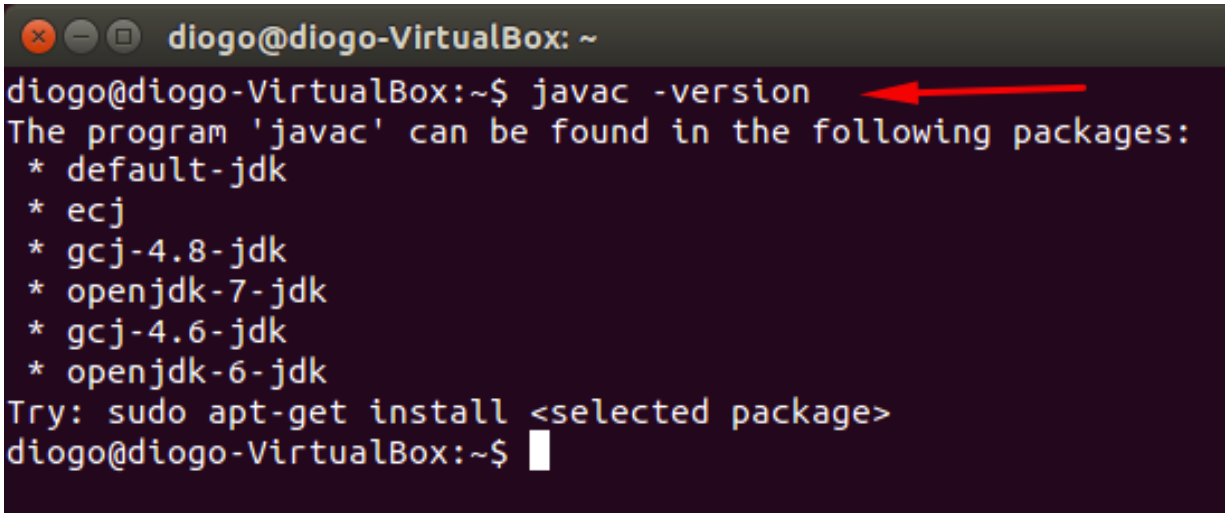


Figura 9.1: Linux sem o Java instalado

Se a saída for algo como a imagem anterior, não temos o JDK instalado, então precisamos baixá-lo e configurá-lo. Se você já possui o JDK instalado, o terminal vai retornar a versão do `javac`, que é o binário utilizado pelo JDK, algo como `javac 1.8.0_91`.

Caso você já possua o Oracle JDK 7 ou superior, desconsidere os passos 1, 2 e 3 da instalação.

> **ORACLE JDK VERSUS OPENJDK**
>
> A documentação oficial do Cordova orienta instalar o Oracle JDK 7 ou posterior.

Vamos à instalação:

1. Acesse o site http://bit.ly/oraclejavaJDK, e escolha a opção de download do *Java Platform* (JDK).
2. Selecione o arquivo para Linux (x86 ou x64) no formato `.tar.gz`.
3. Descompacte em `/usr/lib/jvm/` ou em um diretório de sua preferência (basta lembrar o caminho).

4. Adicione ao `PATH` do sistema a variável `JAVA_HOME` e também o caminho da JDK. Abra o arquivo `~/.bash_profile`:

   ```
   nano ~/.bash_profile
   ```

   Copie e cole os comandos a seguir, lembrando do caminho do diretório da etapa anterior. Atente-se também ao fato de que aqui o diretório extraído do Oracle JDK chama-se `jdk1.8.0_91`, mas o seu certamente será diferente devido às atualizações da Oracle:

   ```
   export JAVA_HOME=/usr/lib/jvm/jdk1.8.0_91
   export PATH=$PATH:/usr/lib/jvm/jdk1.8.0_91/bin
   ```

5. Recarregue as configurações do arquivo:

   ```
   source ~/.bash_profile
   ```

6. (Opcional) Caso você não tenha o `javac` disponível ainda, o comando a seguir cria um link simbólico para deixá-lo disponível:

   ```
   sudo update-alternatives --install "/usr/bin/javac" "javac"
   "/usr/lib/jvm/jdk1.8.0_91/bin/javac" 1
   ```

7. (Opcional) O comando seguinte configura que o `javac` seja respondido pelo Oracle JDK. Ele é utilizado quando você já tem outras distribuições do Java JDK instaladas.

   ```
   sudo update-alternatives --set javac
   /usr/lib/jvm/jdk1.8.0_91/bin/javac
   ```

Finalizado a instalação, vamos verificar se tudo está correto com: `javac -version`. Se a saída for como a da figura adiante e retornar a versão, então podemos prosseguir; caso contrário, refaça os passos anteriores e veja se não pulou alguma etapa.

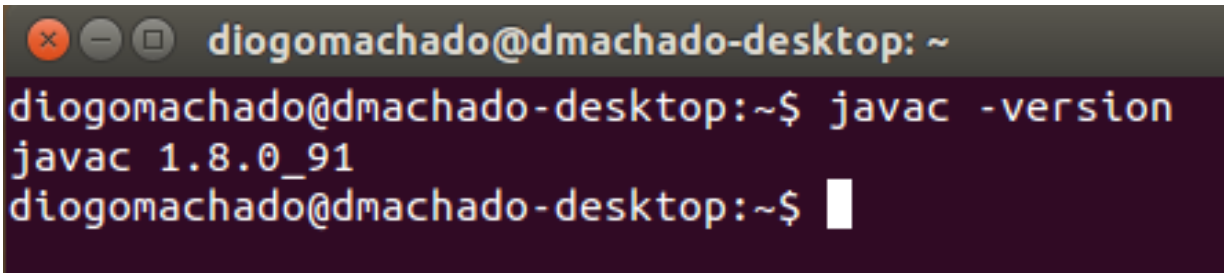


Figura 9.2: Linux com o Java JDK instalado

Para o Java JDK, essas configurações são suficientes. Vamos adiante com a configuração do Android SDK.

> **Dica**
>
> Se a variável do sistema `JAVA_HOME` não estiver disponível usando o comando `echo $JAVA_HOME` quando você abrir um novo terminal, adicione a linha seguinte ao arquivo `.bashrc`, localizado em `~/.bashrc`. Ela vai garantir o carregamento do `.bash_profile` sempre que você abrir um terminal:
>
> ```
> . ~/.bash_profile
> ```

Configurando o Android SDK

Acesse http://developer.android.com/sdk, para ver um link para baixar o Android Studio e o SDK. Escolhemos essa opção porque queremos ter o Android Studio instalado, já que pode ser útil para algumas tarefas e customizações, como também porque é o IDE oficial para quem desenvolve para Android. Feito o download, descompacte o arquivo `.zip` no diretório `home` do seu usuário.

Dentro do diretório do Android Studio, existe um *script* para iniciar o IDE. Considerando que o diretório que você descompactou o IDE é o caminho `~/android-studio`, basta rodar no terminal:

```
.~/android-studio/bin/studio.sh
```

Depois de aberto, siga avançando. O Android Studio vai pedir a instalação de alguns pacotes do SDK, como a figura a seguir:

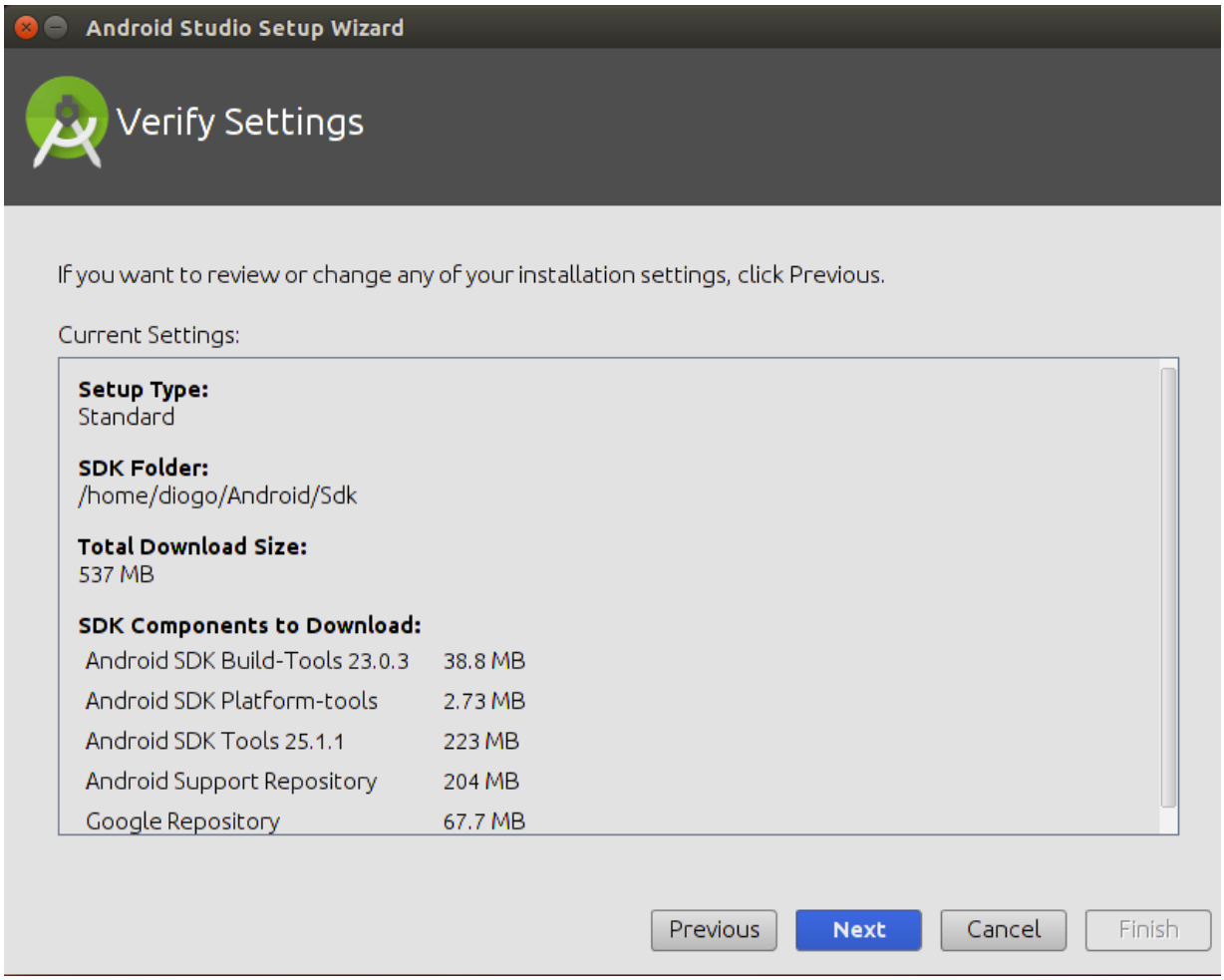


Figura 9.3: Download da SDK via Android Studio

ANDROID STUDIO EM DISTRIBUIÇÕES 64 BITS

Se você usa uma distribuição Linux 64 bits, precisará de algumas bibliotecas 32 bits. Para instalá-las, execute o seguinte comando no terminal:

```
sudo apt-get install lib32z1 lib32ncurses5 lib32bz2-1.0 lib32stdc++6
```

Android Studio no Fedora 64 bits

Se você usa a distribuição Fedora 64 bits, você precisa de algumas bibliotecas específicas também; execute o seguinte comando no terminal:

```
sudo apt-get install lib32z1 lib32ncurses5 lib32bz2-1.0 lib32stdc++6
```

Adicionando os diretórios do Android SDK ao PATH do sistema operacional

Instalado o SDK, é hora de adicionar ao `PATH` do sistema alguns caminhos do Android SDK. Abra o terminal, crie ou edite o arquivo `~/.bash_profile` .

Para criar o `~/.bash_profile` , faça:

```
touch ~/.bash_profile
```

Para editar o `~/.bash_profile` , faça:

```
nano ~/.bash_profile
```

Insira os dois comandos `export` dentro do arquivo `.bash_profile` . Essas duas linhas vão tornar os comandos do Android SDK disponíveis no terminal:

```
export ANDROID_HOME=~/android-studio
export PATH=${PATH}:${ANDROID_HOME}/platform-tools:${ANDROID_HOME}/tools
```

Recarregue as configurações do arquivo:

```
source ~/.bash_profile
```

Feito isso, feche e abra o terminal novamente. Então, você terá o comando `android` disponível.

SHELL CUSTOMIZADO (OH MY ZSH)

Se você usa o shell customizado do http://ohmyz.sh, abra o arquivo `~/.zshrc` e adicione a linha a seguir ao seu final:

```
if [ -f ~/.bash_profile ]; then . ~/.bash_profile; fi
```

Você pode testar, executando o seguinte comando: `android` . Se tudo deu certo, ele abrirá o Android SDK Manager como a imagem a seguir. Este é o gerenciador de pacotes no qual você fará todos os downloads das versões do Android, ferramentas e dependências. Também é nele que criamos os AVDs, os *dispositivos virtuais Android*, também conhecidos como emuladores.

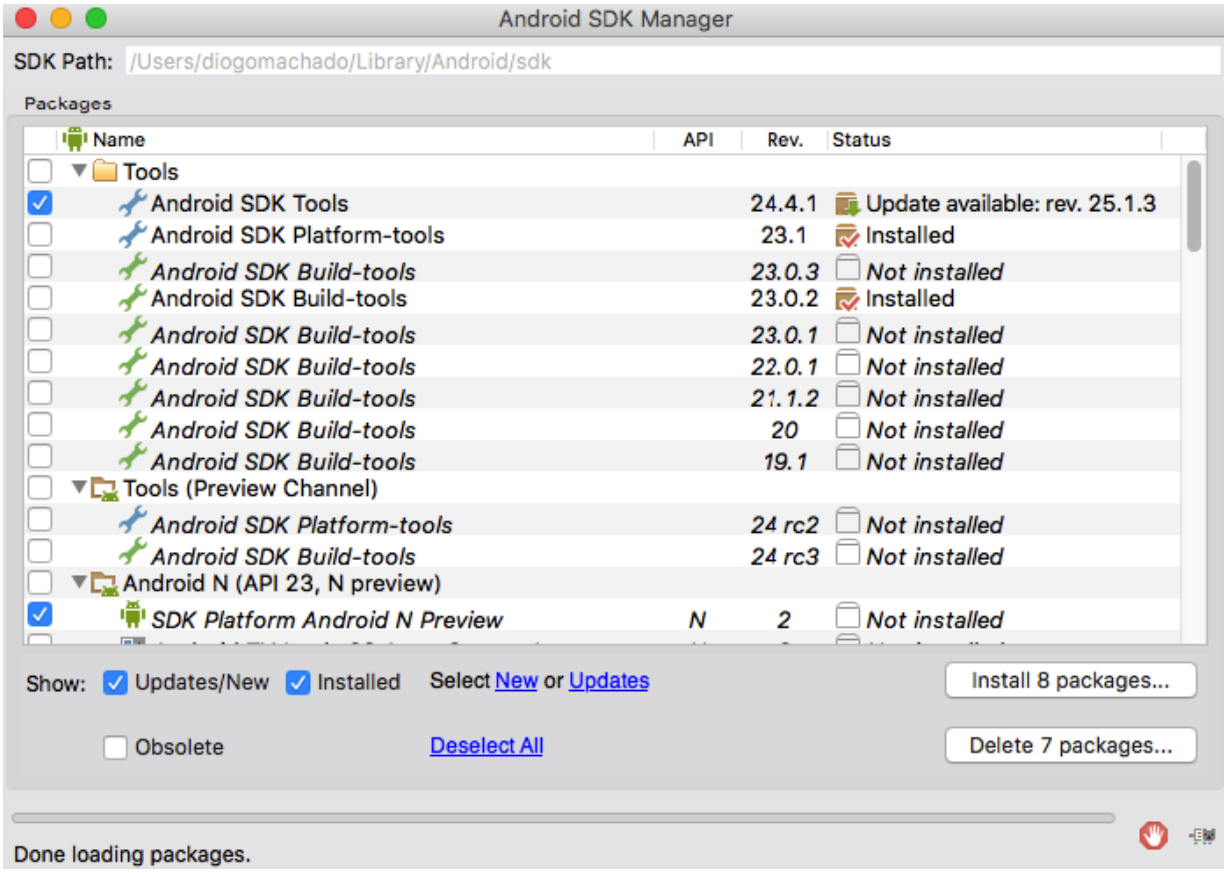

Android SDK Manager
SDK Path: /Users/diogomachado/Library/Android/sdk
Packages
Name
API
Rev.
Status
Tools
Android SDK Tools 24.4.1 Update available: rev. 25.1.3
Android SDK Platform-tools 23.1 Installed
Android SDK Build-tools 23.0.3 Not installed
Android SDK Build-tools 23.0.2 Installed
Android SDK Build-tools 23.0.1 Not installed
Android SDK Build-tools 22.0.1 Not installed
Android SDK Build-tools 21.1.2 Not installed
Android SDK Build-tools 20 Not installed
Android SDK Build-tools 19.1 Not installed
Tools (Preview Channel)
Android SDK Platform-tools 24 rc2 Not installed
Android SDK Build-tools 24 rc3 Not installed
Android N (API 23, N preview)
SDK Platform Android N Preview N 2 Not installed
Show: Updates/New Installed Select New or Updates
Install 8 packages...
Obsolete
Deselect All
Delete 7 packages...
Done loading packages.

Figura 9.4: Android SDK Manager

Caso o comando `android` não abra o gerenciador, volte e reveja se fez todos os passos corretamente.

Faça o download inicialmente dos pacotes:

- **Android SDK Tools** – Componente com as ferramentas essenciais para emulação no Android;
- **Android SDK Platform-tools** – Inclui várias ferramentas requeridas para a plataforma Android, como o *adb (Android Debug Bridge)*;
- **Android SDK Build-tools** – É um componente necessário que inclui ferramentas para o *build* das aplicações no Android;
- **API Level** – Baixe a última versão da API Level do Android, geralmente descrito no gerenciador com o padrão *nome da versão* mais *número API*, por exemplo, **Android 6 (API 23)**. É **importantíssimo** que você baixe a última versão mesmo que não queira trabalhar com ela, porque o CLI do Cordova tenta acompanhar sempre a última versão do API Level.

> **API Level e o Cordova**
>
> Se você está com dúvidas quanto ao Cordova e ao API Level necessário, acesse a documentação oficial: http://cordova.apache.org/docs/en/latest/guide/platforms/android/index.html.

Baixe e instale o Apache Ant

Apache Ant é uma ferramenta usada para automação de compilação na construção de software na linguagem Java. Ela é uma biblioteca **indispensável** para o Android SDK fazer a construção (*build*) da aplicação Android, e utiliza um arquivo XML para descrever o processo de construção – por padrão, chamado `build.xml` .

Acesse http://ant.apache.org, vá à seção `Download* -> Binary Distributions` e, em `Current Release of Ant`, baixe o arquivo no formato `.zip`. Feito o download, faça o *unzip* no diretório `home` do seu usuário. Agora, abra o arquivo que criamos anteriormente chamado `.bash_profile` no terminal:

```
nano ~/.bash_profile
```

Insira uma linha com o caminho do diretório do Ant:

```
export PATH=$PATH:~/apache-ant-1.9.7/bin
```

Salve e recarregue as configurações do arquivo:

```
source ~/.bash_profile
```

Instale o Node.js

Agora vamos instalar o Node.js, necessário para instalarmos o pacote do CLI do Cordova e do PhoneGap. O Node.js também abre portas para outros pacotes disponibilizados no https://www.npmjs.com, que podem ser muito úteis no desenvolvimento híbrido (como o Gulp e o Browser-sync).

Vamos à instalação. Abra o terminal e execute os comandos:

```
sudo apt-get update
sudo apt-get install nodejs
```

Na distribuição Ubuntu do Linux, diferente de outros sistemas, o Node.js é instalado com o atalho chamado `nodejs` – por padrão, nos outros sistemas é `node`.

```
nodejs -v
```

Se o comando retornar a versão do node, sua instalação está correta; se não retornar, tente repetir os passos.

Instalando os pacotes do CLI do Cordova e do PhoneGap

Agora que temos o NPM disponível, podemos instalar o CLI do Cordova e o do PhoneGap:

```
sudo npm install -g cordova
```

Em seguida, instalaremos o PhoneGap também:

```
sudo npm install -g phonegap
```

Para completar nossa configuração, precisamos instalar os pacotes a seguir, que são necessários para o comando `build` no Linux:

```
sudo apt-get install lib32stdc++6 lib32z1
sudo apt-get install android-tools-adb
```

Fechamos por aqui a configuração do Android no Linux, então, vamos relembrar o que fizemos:

- Começamos verificando se tínhamos o Java; se não, instalamos.
- Definimos a variável de ambiente chamada `JAVA_HOME`.
- Baixamos e instalamos o Android SDK.
- Adicionamos ao `PATH` do sistema os caminhos necessários do Android SDK e também a variável `ANDROID_HOME`.
- Com o *Android SDK Manager* aberto, baixamos alguns pacotes mínimos para posteriormente fazer o `build` da aplicação.
- Baixamos a biblioteca Apache Ant.
- Adicionamos o diretório do Apache Ant ao `PATH`.
- Instalamos o Node.js.
- Instalamos o CLI do Cordova e do PhoneGap.
- Por fim, instalamos algumas bibliotecas de dependências necessárias.

9.2 Android no macOS

O sistema operacional Android é desenvolvido em cima de uma API estruturada na linguagem Java, por isso, a documentação do Cordova recomenda instalar o Oracle Java JDK (*Java Development Kit*). Ele será utilizado pelo Android SDK (*kit de desenvolvimento de software*) a todo momento.

Verifique se sua máquina já possui o `javac` (compilador da linguagem Java), disponível quando já temos o JDK instalado. Para isso, abra o terminal do Linux e rode o seguinte comando:

```
javac -version
```

Se o comando não retornar o diretório do `javac` , não temos o JDK instalado, então precisamos baixá-lo e configurá-lo. Se você já o possui, o terminal vai retornar a versão do `javac` , que é o binário utilizado pelo JDK, algo como `javac 1.8.0_91` .

Caso você já possua o Oracle JDK 7 ou superior, desconsidere os passos 1, 2 e 3 da instalação.

> **ORACLE JDK VERSUS OPENJDK**
>
> A documentação oficial do Cordova orienta a instalar o Oracle JDK 7 ou posterior.

Acesse o site da Oracle e baixe o JDK: http://www.oracle.com/technetwork/pt/java/javase/downloads/index.html.

1. Clique no link:

2. Escolha a versão `.dmg` para macOS:

Java SE Development Kit 8u91

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

○ Accept License Agreement ◉ Decline License Agreement

Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	77.72 MB	jdk-8u91-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	74.69 MB	jdk-8u91-linux-arm64-vfp-hflt.tar.gz
Linux x86	154.74 MB	jdk-8u91-linux-i586.rpm
Linux x86	174.92 MB	jdk-8u91-linux-i586.tar.gz
Linux x64	152.74 MB	jdk-8u91-linux-x64.rpm
Linux x64	172.97 MB	jdk-8u91-linux-x64.tar.gz
Mac OS X	227.29 MB	jdk-8u91-macosx-x64.dmg
Solaris SPARC 64-bit (SVR4 package)	139.59 MB	jdk-8u91-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	98.95 MB	jdk-8u91-solaris-sparcv9.tar.gz
Solaris x64 (SVR4 package)	140.29 MB	jdk-8u91-solaris-x64.tar.Z
Solaris x64	96.78 MB	jdk-8u91-solaris-x64.tar.gz
Windows x86	182.11 MB	jdk-8u91-windows-i586.exe
Windows x64	187.41 MB	jdk-8u91-windows-x64.exe

3. Feito o download, execute o instalador e vá avançando até finalizar.

Para o Java, essas configurações são suficientes. Vamos ao Android SDK.

Configurando o Android SDK

Acesse http://developer.android.com/sdk, para ver um link para baixar o Android Studio e o SDK. Escolhemos essa opção porque queremos ter o Android Studio instalado, útil para algumas tarefas e customizações, como também por ser o IDE oficial para quem desenvolve para Android.

Abra o instalador e siga avançando. O Android Studio pedirá a instalação de alguns pacotes do SDK, como na figura a seguir:

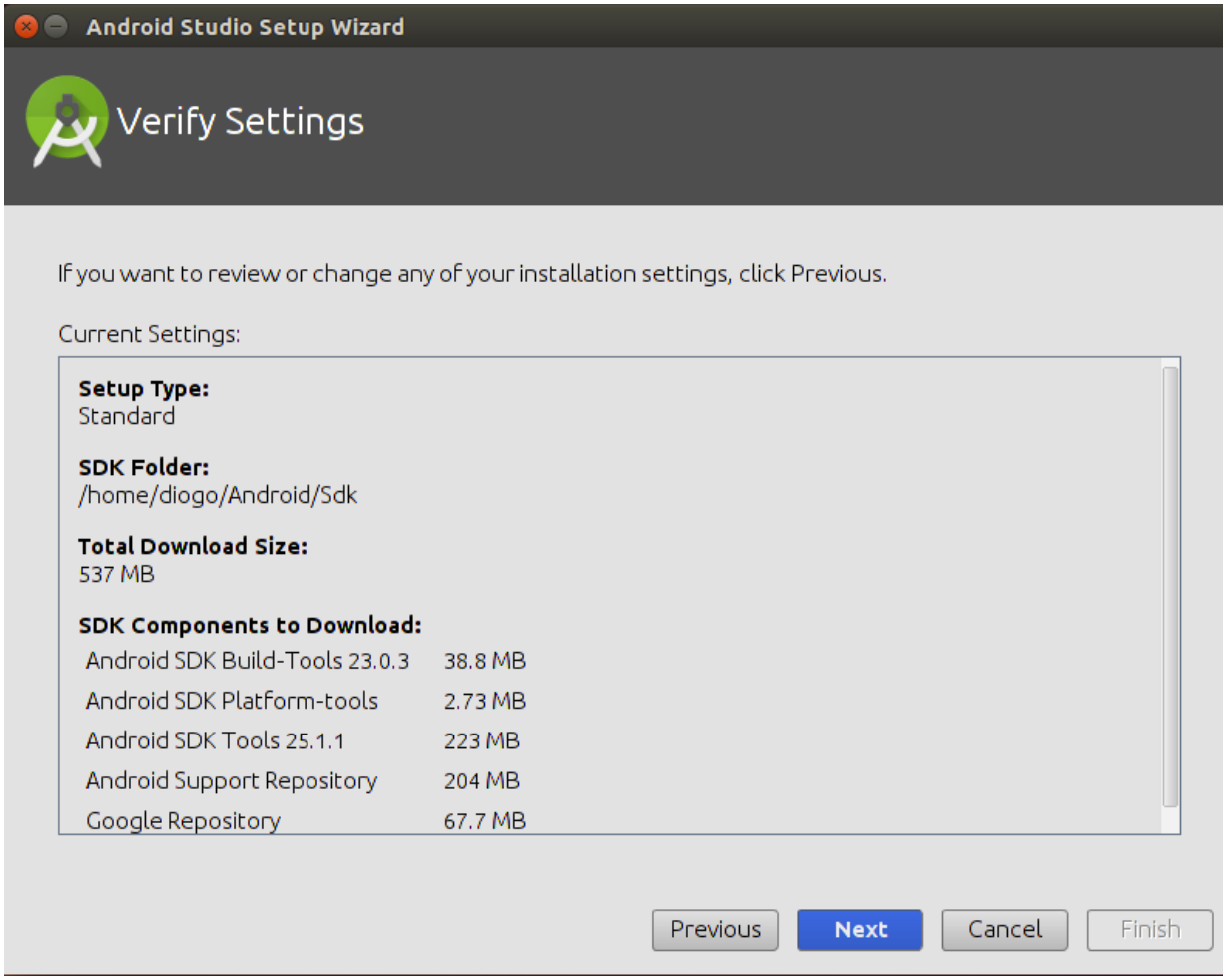


Figura 9.7: Download da SDK via Android Studio

Configure as variáveis do sistema

Instalado o SDK, agora precisamos configurar algumas **variáveis do sistema** para as termos disponíveis no terminal. Abra o terminal,

crie ou edite o arquivo `~/.bash_profile` .

Para criar, faremos:

```
touch ~/.bash_profile
```

Para editar, faremos:

```
nano ~/.bash_profile
```

Agora você vai colar o `export` a seguir dentro do arquivo `~/.bash_profile` . Essa linha tornará os comandos necessários do Android SDK disponíveis no terminal:

```
export PATH=$PATH:~/Library/Android/sdk/platform-
tools/:~/Library/Android/sdk/tools
```

Então, precisamos recarregar as configurações do arquivo:

```
source ~/.bash_profile
```

Feito isso, feche e abra o terminal novamente, para ter os comandos necessários do Android disponíveis. Você pode testar, executando o seguinte comando:

```
android
```

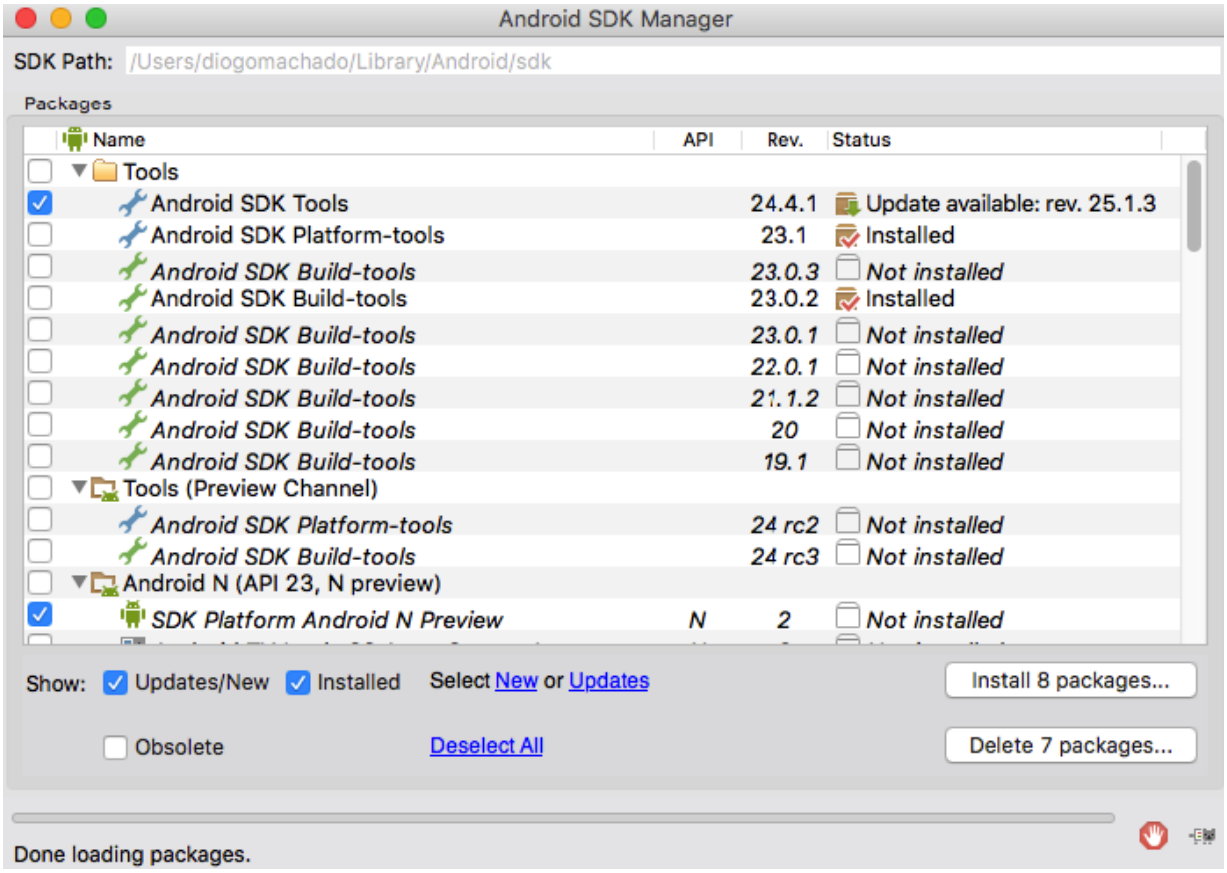


Figura 9.8: Android SDK Manager aberto

Faça o download inicialmente dos pacotes:

- **Android SDK Tools** – Componente com as ferramentas essenciais para emulação no Android.
- **Android SDK Platform-tools** – Inclui várias ferramentas requeridas para a plataforma Android, como o adb *Android Debug Bridge* (ADB).
- **Android SDK Build-tools** – É um componente necessário que inclui ferramentas para o *build* das aplicações no Android.
- **API Level** – Baixe a última versão da API Level do Android, geralmente descrito no gerenciador com o padrão *nome da versão* mais *número API*, como **Android 6 (API 23)**. É **importantíssimo** que você baixe a última versão mesmo que não queira trabalhar com ela, porque o CLI do Cordova tenta acompanhar sempre a última versão do API Level.

Baixe e instale o Apache Ant

Apache Ant é uma ferramenta utilizada para automação de compilação na construção de software na linguagem Java. É uma biblioteca **indispensável** para o Android SDK fazer a construção (*build*) da aplicação Android. O Ant utiliza um arquivo XML para descrever o processo de construção – por padrão, chamado `build.xml` .

Acesse http://ant.apache.org, vá à seção `Download -> Binary Distributions` e, em `Current Release of Ant` , você pode baixar o arquivo no formato `.zip` . Feito o download, *unzip* esse arquivo no diretório `home` do seu usuário.

Agora, abra o arquivo que criamos anteriormente `.bash_profile` no terminal:

```
nano ~/.bash_profile
```

Insira uma linha com o caminho do diretório do Apache Ant ao `PATH` :

```
export PATH=$PATH:~/apache-ant-1.9.7/bin
```

Salve e recarregue as configurações do arquivo:

```
source ~/.bash_profile
```

Instale o Node.js

Agora vamos instalar o Node.js, necessário para instalarmos a linha de comando (CLI) tanto do Cordova quanto do PhoneGap, como também o Node.js, que abre portas para outros pacotes disponibilizados no https://www.npmjs.com que podem ser muito úteis no desenvolvimento híbrido (como o Gulp e o Browser-sync).

Acesse o site https://nodejs.org, baixe o instalador `.dmg` na sua versão LTS e, depois, é só ir avançando.

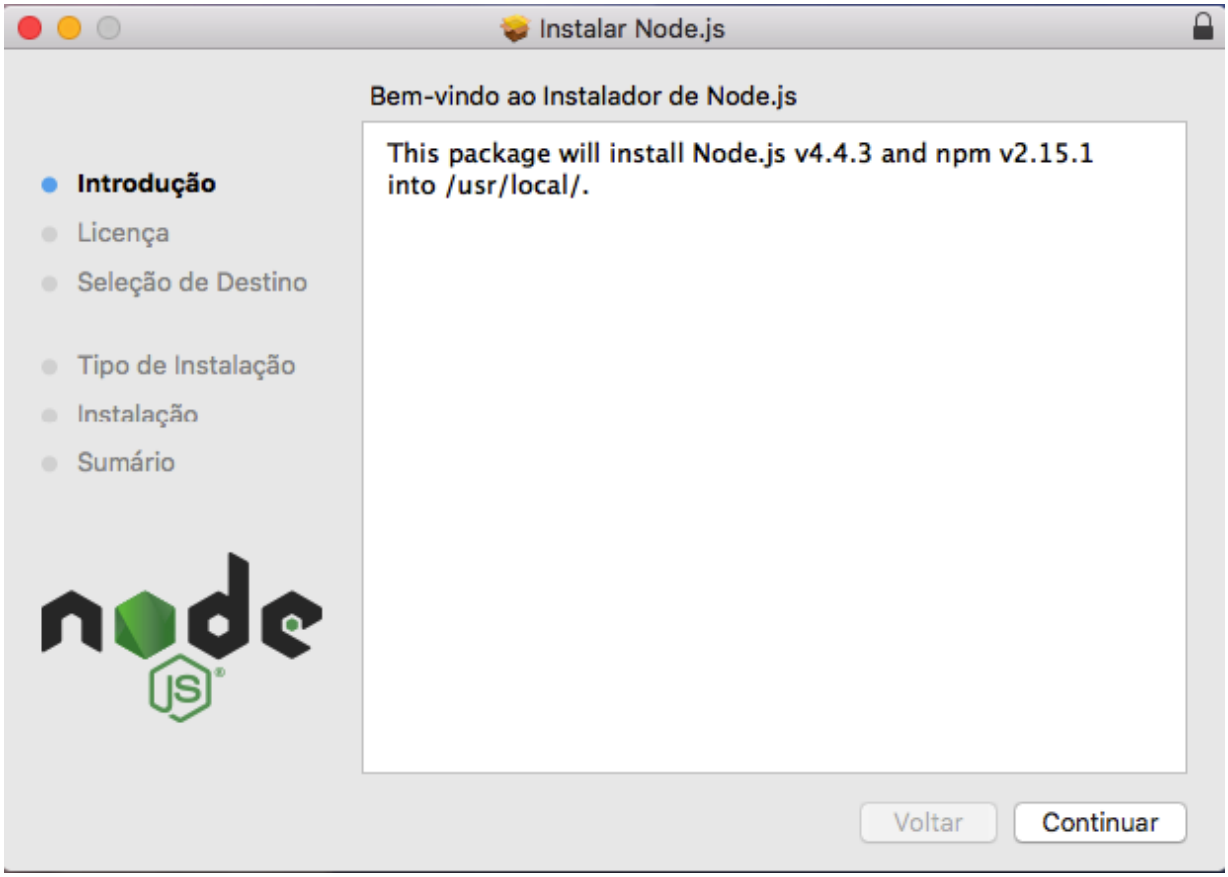


Figura 9.9: Instalação do Node.js no macOS

Como mostrado na figura anterior, o instalador já vem com o Node.js, o NPM e configura o `PATH` do sistema.

Instale o CLI do Cordova e do PhoneGap

Agora que temos o NPM disponível, podemos instalar o CLI do Cordova e o do PhoneGap:

```
sudo npm install -g cordova
```

Em seguida, instalaremos o PhoneGap também:

```
sudo npm install -g phonegap
```

Fechamos por aqui a configuração do Android no Mac macOS. Vamos relembrar o que fizemos:

- Começamos verificando se tínhamos o Java; se não, instalamos.
- Baixamos e instalamos o Android SDK com o Android Studio.
- Adicionamos ao `PATH` do sistema os caminhos necessários do Android SDK.
- Com o *Android SDK Manager* aberto, baixamos alguns pacotes mínimos para posteriormente fazer o `build` da aplicação.
- Baixamos a biblioteca Apache Ant.
- Adicionamos o diretório `/bin` do Apache Ant ao `PATH`.
- Instalamos o Node.js.
- Instalamos o CLI do Cordova e do PhoneGap.

9.3 Android no Windows

Como disse, o Android é multiplataforma e também é possível desenvolver para Android no Windows. O sistema operacional Android é desenvolvido em cima de uma API estruturada na linguagem Java, por isso, a documentação do Cordova recomenda instalar o Oracle Java JDK (*Java Development Kit*). Ele será utilizado pelo Android SDK (*kit de desenvolvimento de software*) a todo momento.

Verifique se o Java está instalado

Como o Android é uma API Java, é muito importante termos o Java instalado, porque ele será utilizado pela SDK do Android e pelo Cordova – para fazer o build da aplicação e em diversos processos internos. No terminal, execute o seguinte comando:

```
javac -version
```

Se o comando não retornar a versão do Java, precisamos instalá-lo. Acesse o site da Oracle e baixe o JDK: http://www.oracle.com/technetwork/pt/java/javase/downloads/index.html.

1. Clique no link:

2. Escolha a versão `.exe` e a sua arquitetura (x86 ou x64):

Java SE Development Kit 8u91

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

○ Accept License Agreement ◉ Decline License Agreement

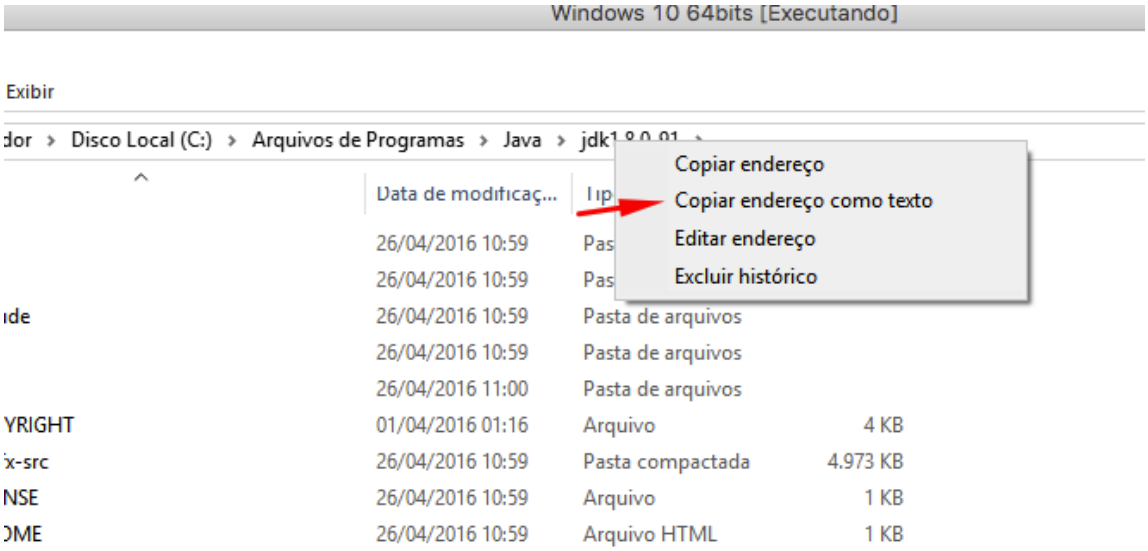
Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	77.72 MB	jdk-8u91-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	74.69 MB	jdk-8u91-linux-arm64-vfp-hflt.tar.gz
Linux x86	154.74 MB	jdk-8u91-linux-i586.rpm
Linux x86	174.92 MB	jdk-8u91-linux-i586.tar.gz
Linux x64	152.74 MB	jdk-8u91-linux-x64.rpm
Linux x64	172.97 MB	jdk-8u91-linux-x64.tar.gz
Mac OS X	227.29 MB	jdk-8u91-macosx-x64.dmg
Solaris SPARC 64-bit (SVR4 package)	139.59 MB	jdk-8u91-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	98.95 MB	jdk-8u91-solaris-sparcv9.tar.gz
Solaris x64 (SVR4 package)	140.29 MB	jdk-8u91-solaris-x64.tar.Z
Solaris x64	96.78 MB	jdk-8u91-solaris-x64.tar.gz
Windows x86	182.11 MB	jdk-8u91-windows-i586.exe
Windows x64	187.41 MB	jdk-8u91-windows-x64.exe

3. Feito o download, execute o instalador e vá avançando até finalizar.

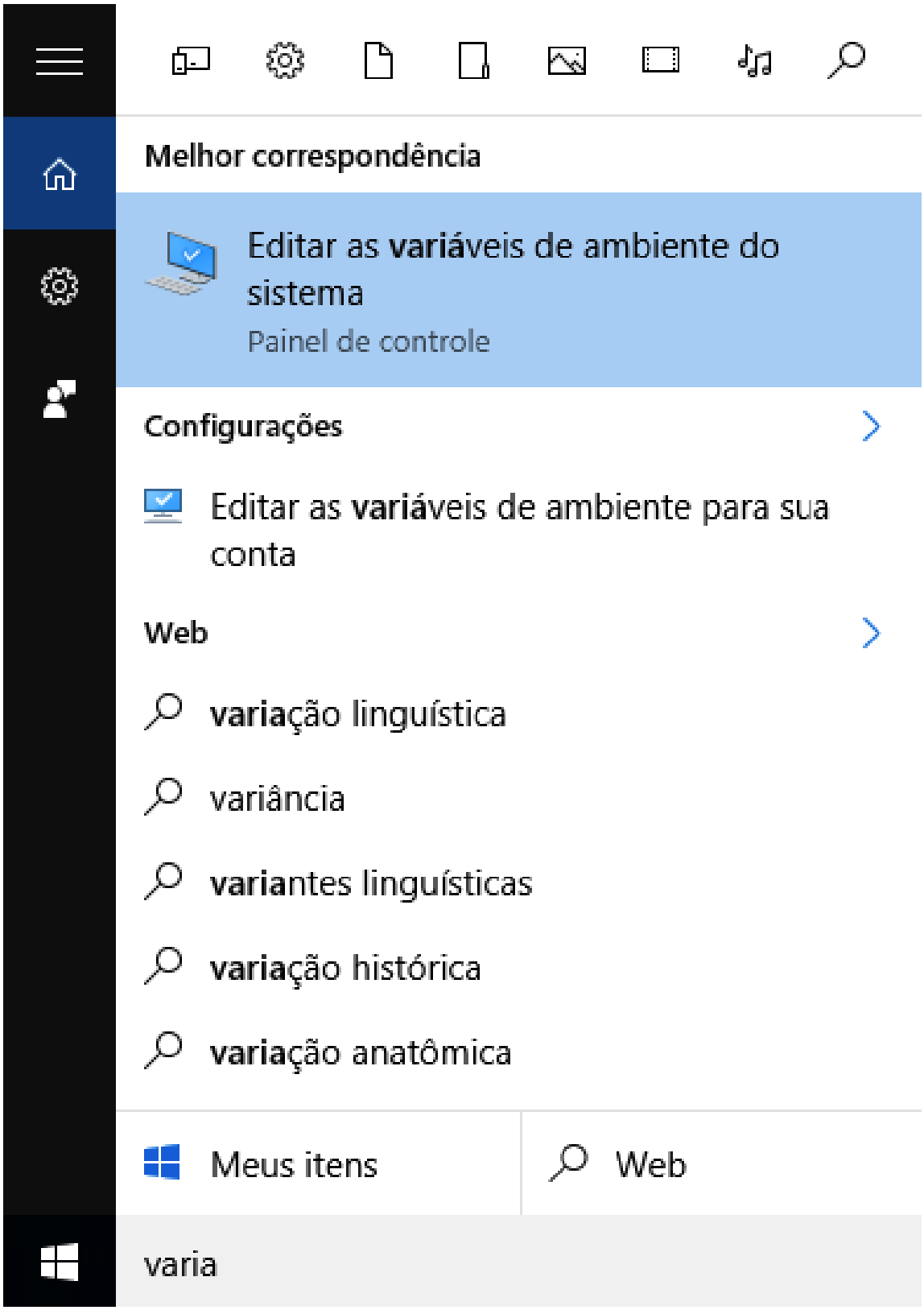
Definindo a variável de ambiente JAVA_HOME

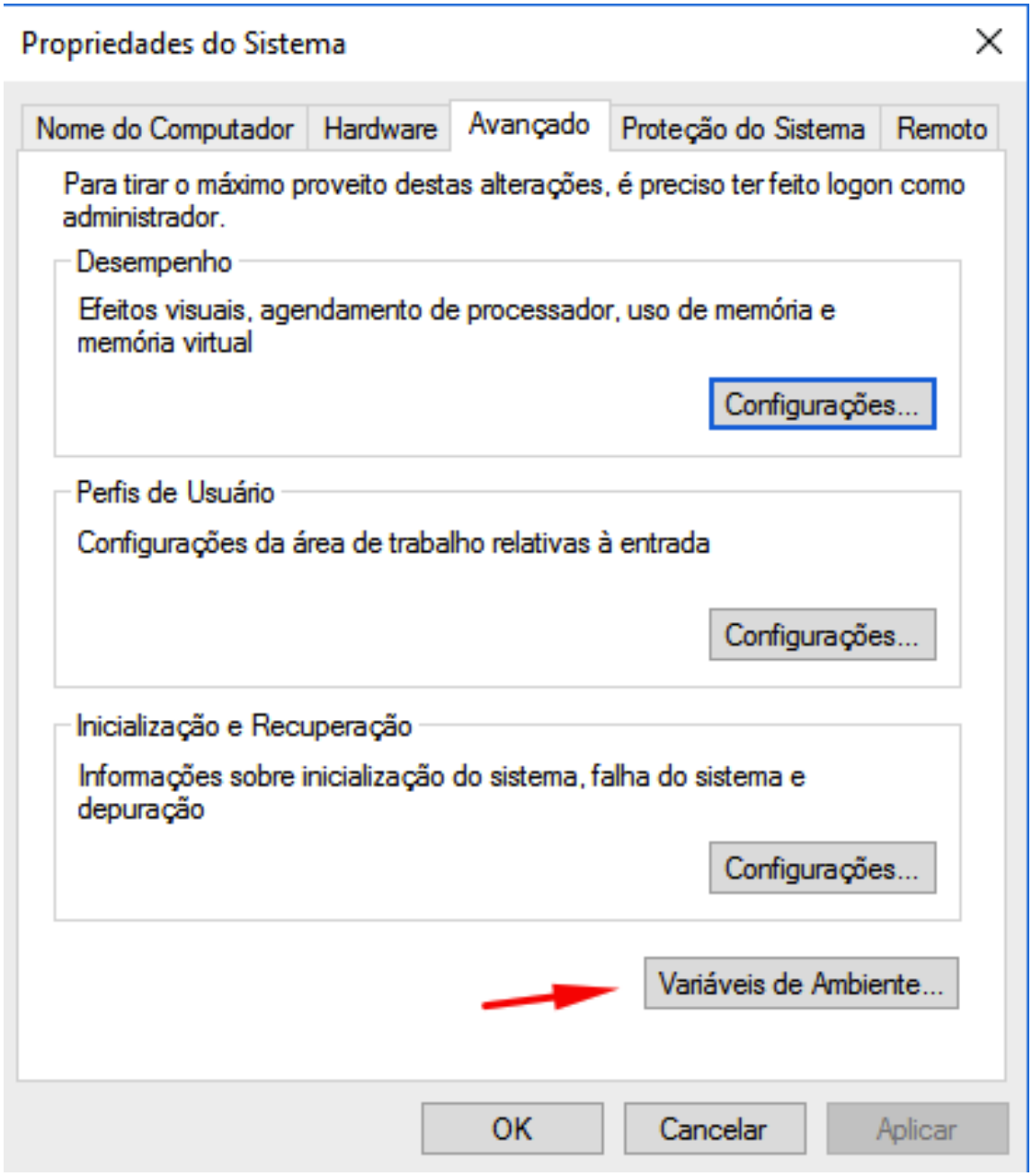
1. Primeiro vamos copiar o endereço do diretório onde está localizado o JDK do Java, normalmente no caminho `C: ->`

`Arquivos de Programa -> Java -> jdk1.8.0.01 .`

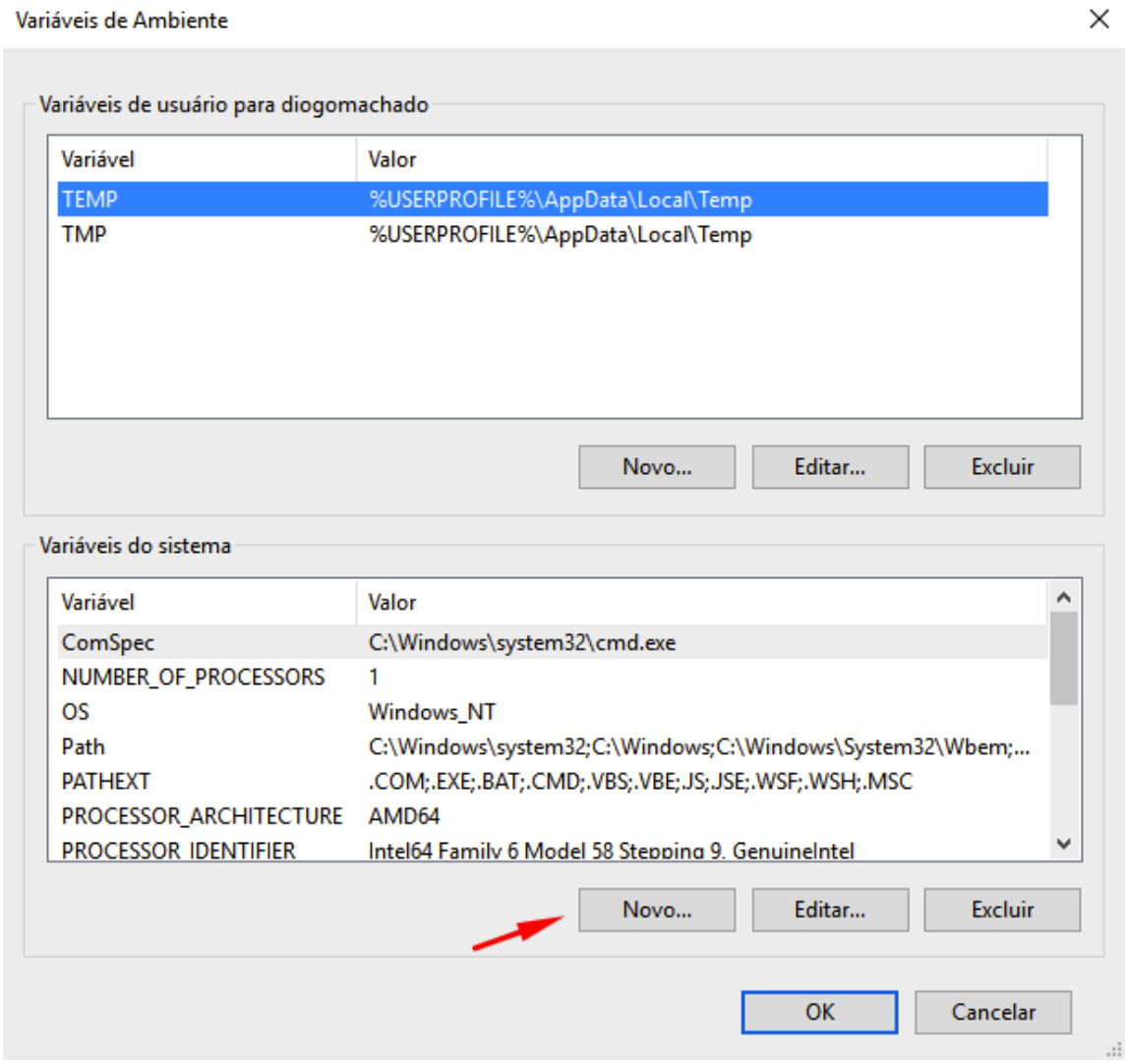


2. Agora, vamos editar as variáveis do sistema.


Melhor correspondência
Editar as variáveis de ambiente do sistema
Painel de controle
Configurações
Editar as variáveis de ambiente para sua conta
Web
variação linguística
variância
variantes linguísticas
variação histórica
variação anatômica
Meus itens
Web
varia

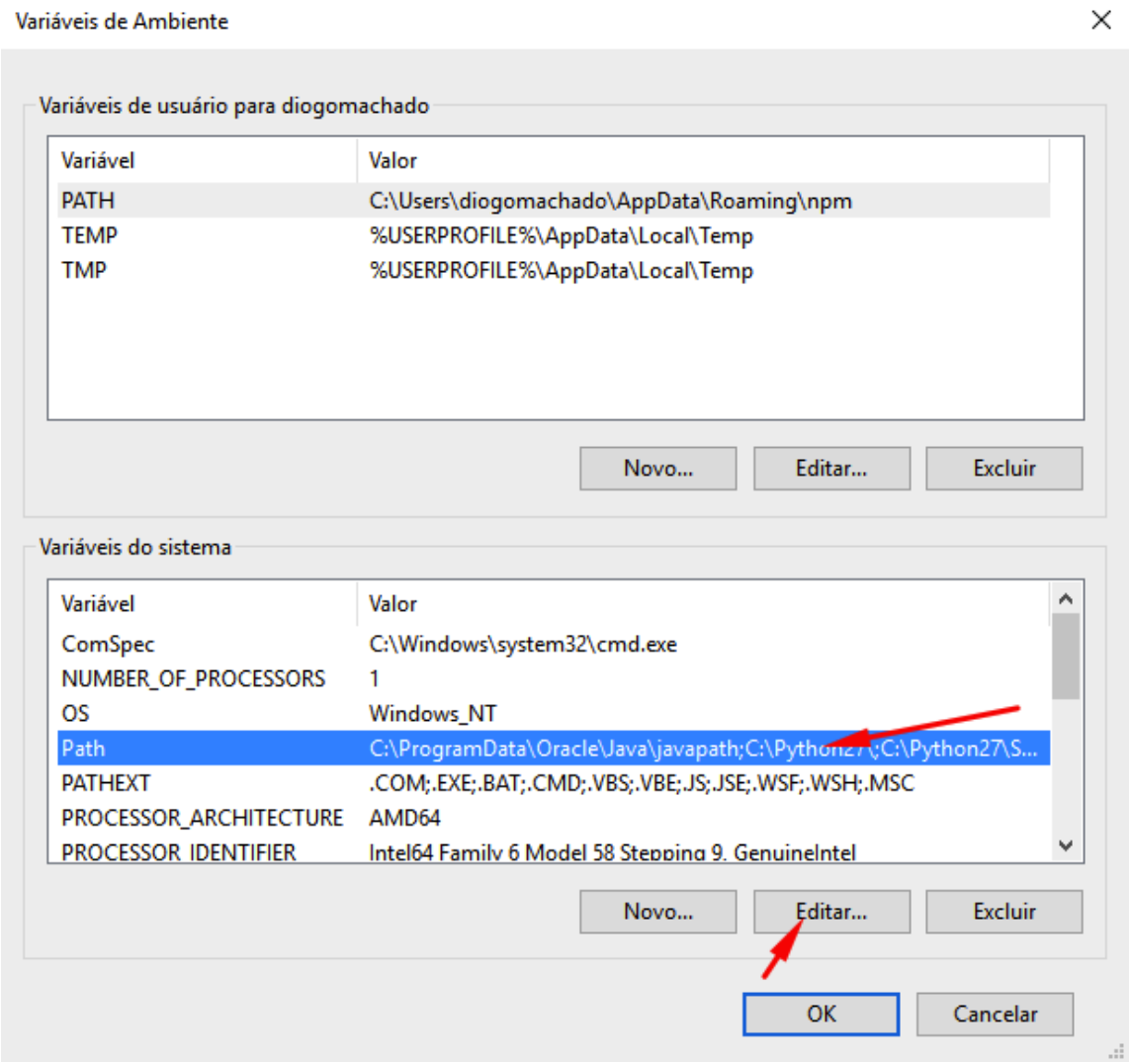


3. Crie uma nova chamada `JAVA_HOME` .




```
![{w=100%}](imagens/c2/windows-java-home.png)
```

1. Adicione também ao `PATH` do sistema.




```
![{w=100%}](imagens/c2/windows-path-edit-java-home.png)
```

Feche e abra o terminal novamente.

Baixe e instale o Android SDK

Acesse http://developer.android.com/sdk, para ver um link para baixar o Android Studio e o SDK. Faça o download e, em seguida, a instalação.

Depois de aberto, você pode ir avançando. Você verá que o Android Studio fará o download do SDK.

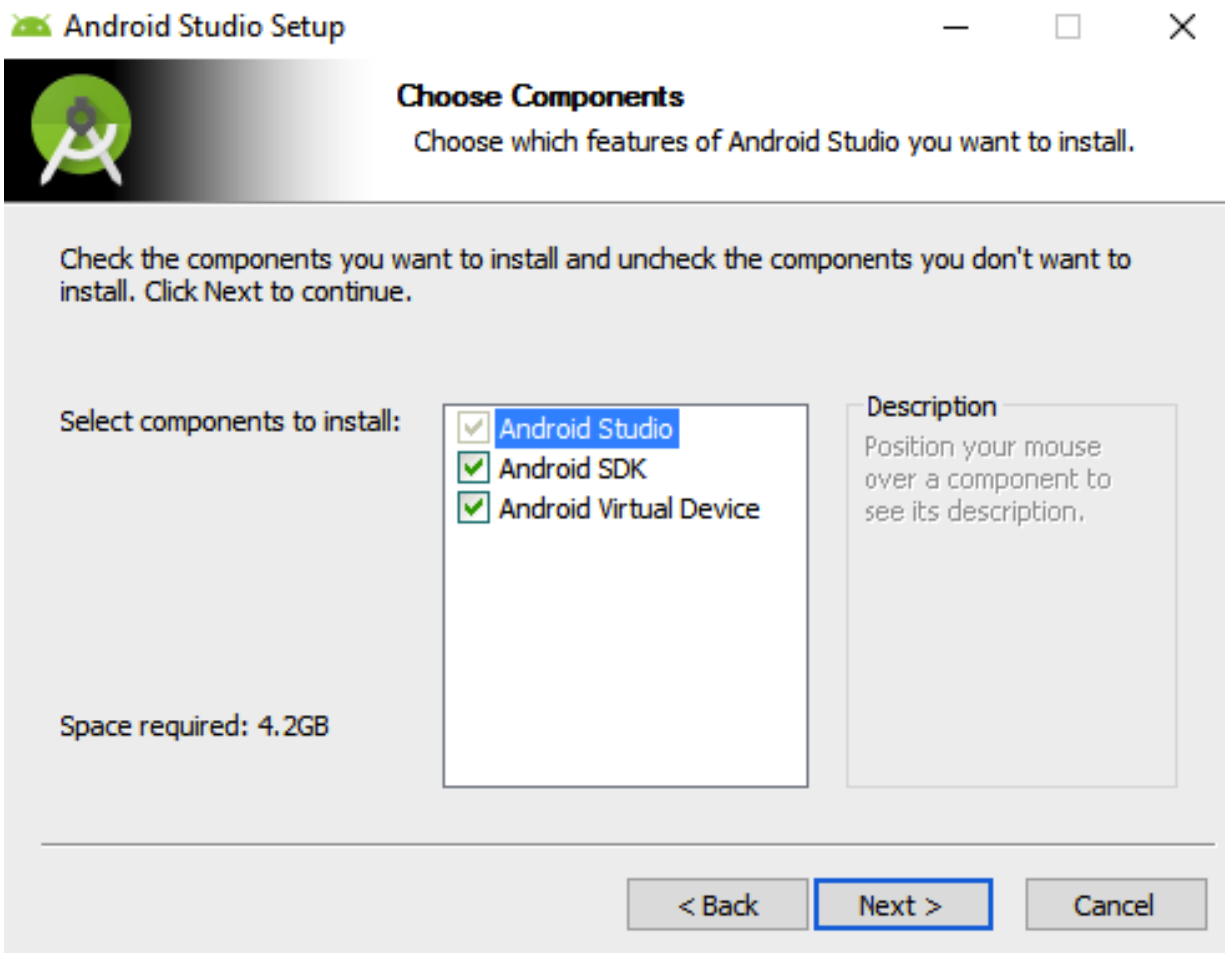

Android Studio Setup
Choose Components
Choose which features of Android Studio you want to install.
Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.
Select components to install:
Android Studio
Android SDK
Android Virtual Device
Description
Position your mouse over a component to see its description.
Space required: 4.2GB
< Back
Next >
Cancel

Figura 9.17: Instalação do Android Studio no Windows 10

Lembre-se do local da instalação do SDK, pois vamos precisar dele em seguida.

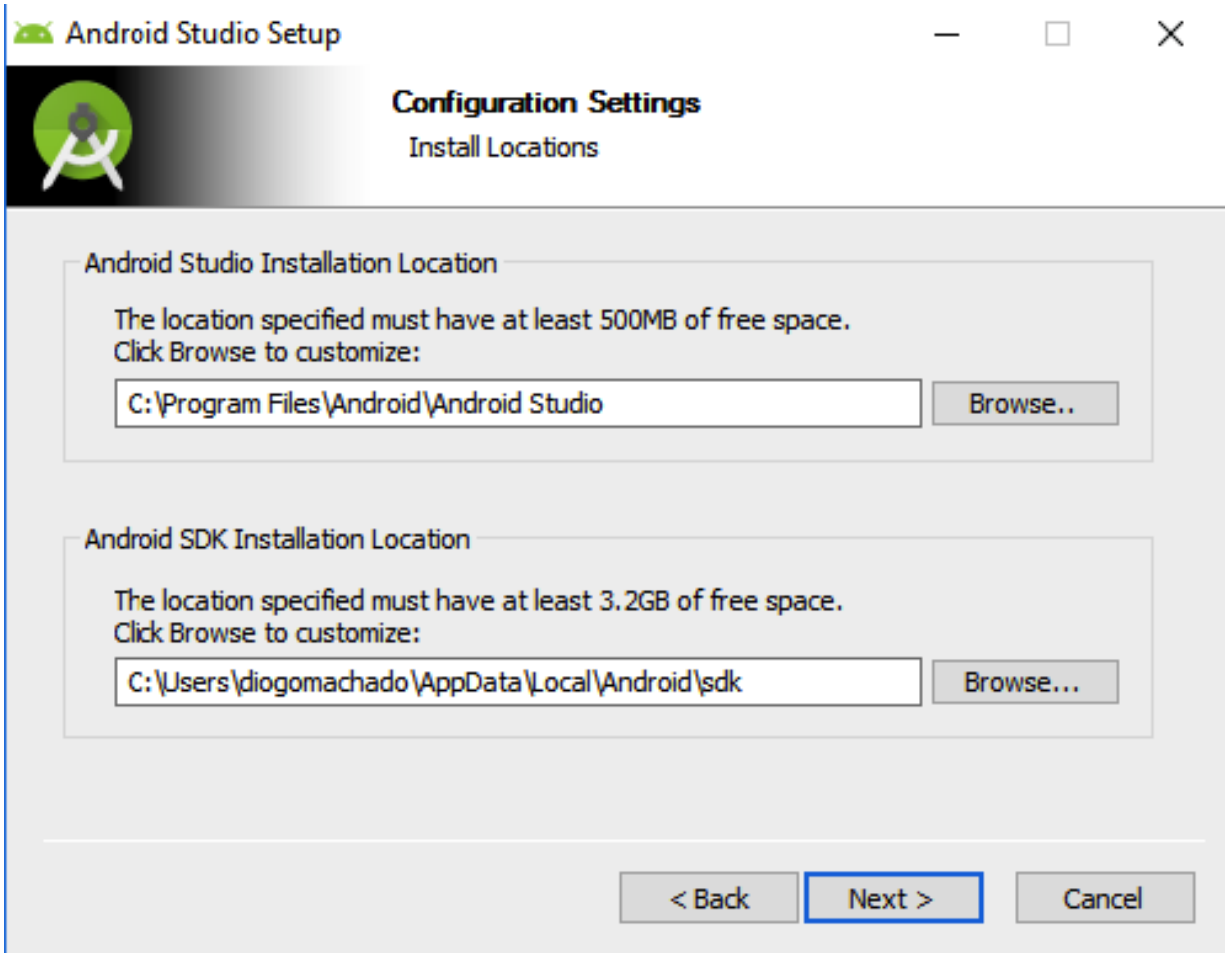
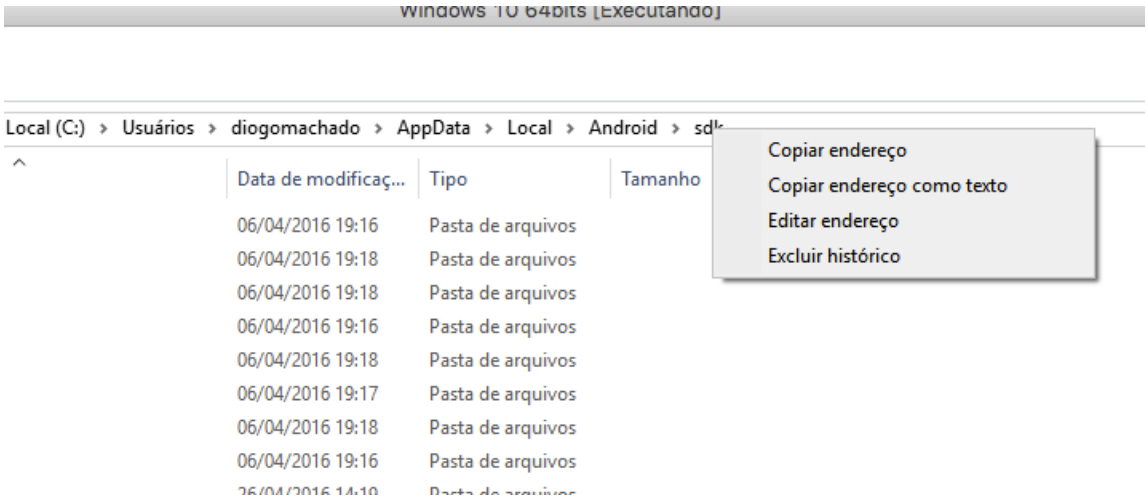


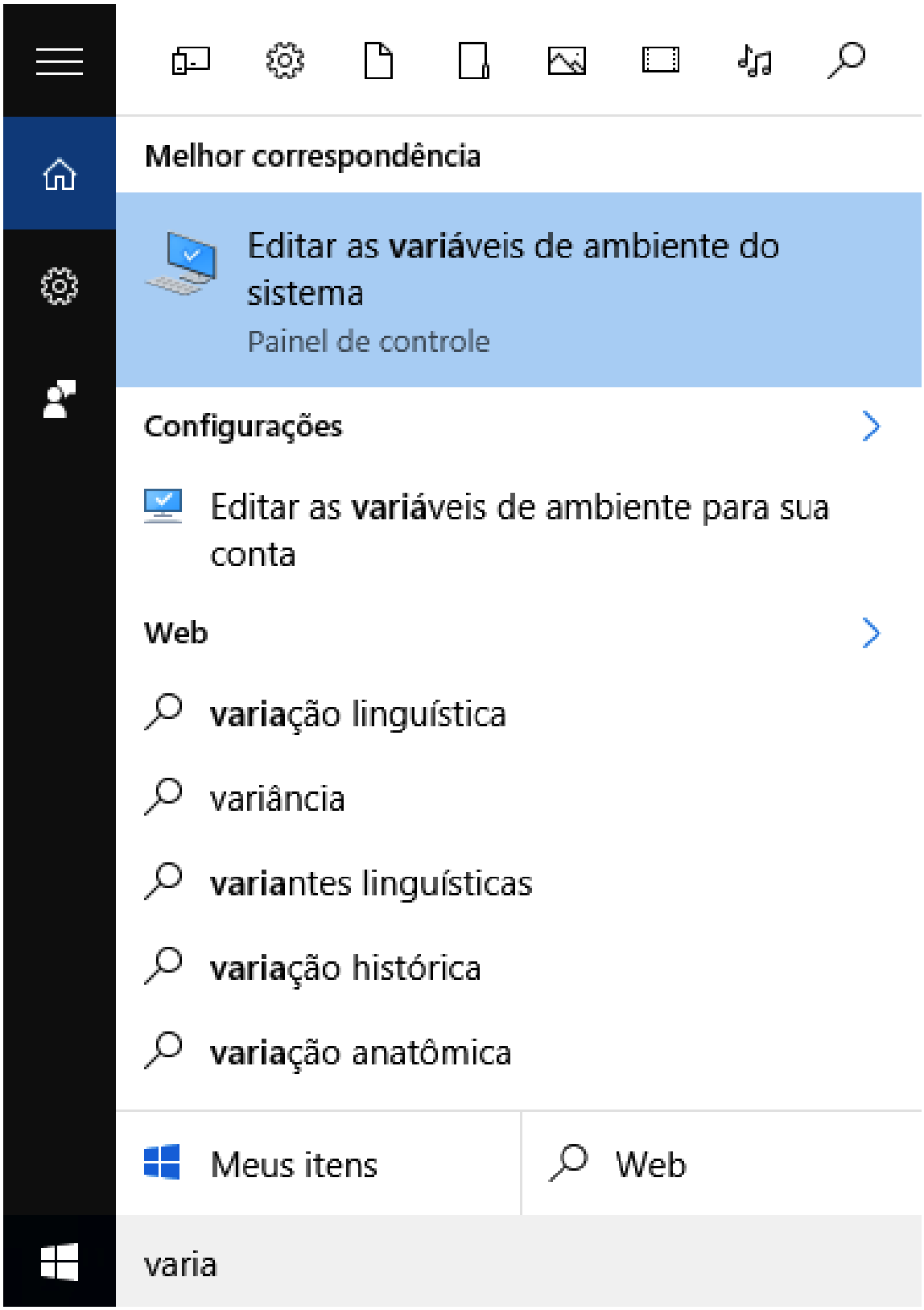
Figura 9.18: Localização do SDK no Windows

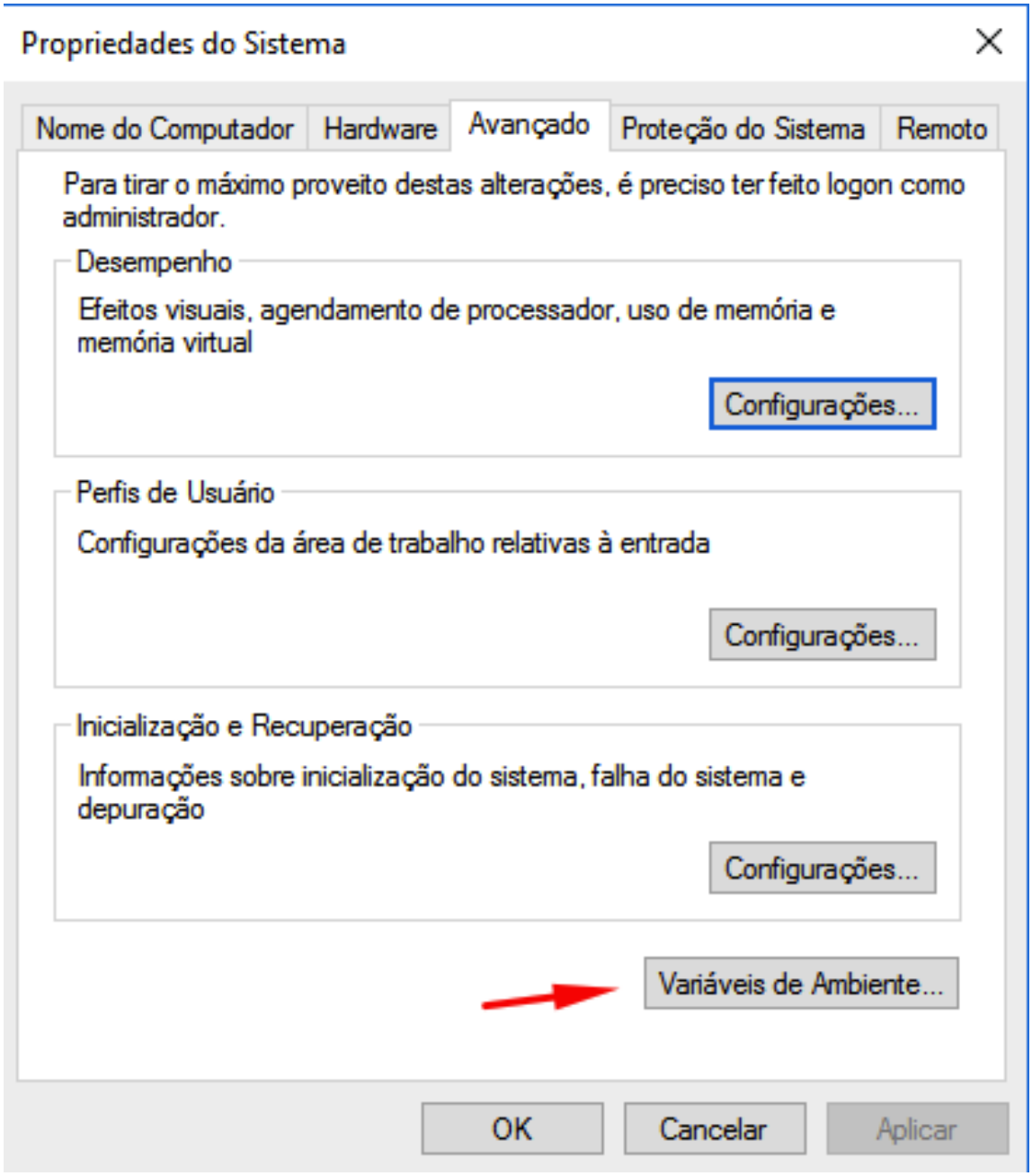
Configure as variáveis do sistema

1. Primeiro vamos copiar o endereço do diretório onde está localizado o SDK. No nosso caso, ele está localizado em `C:\Users\diogomachado\AppData\Local\Android\sdk` .

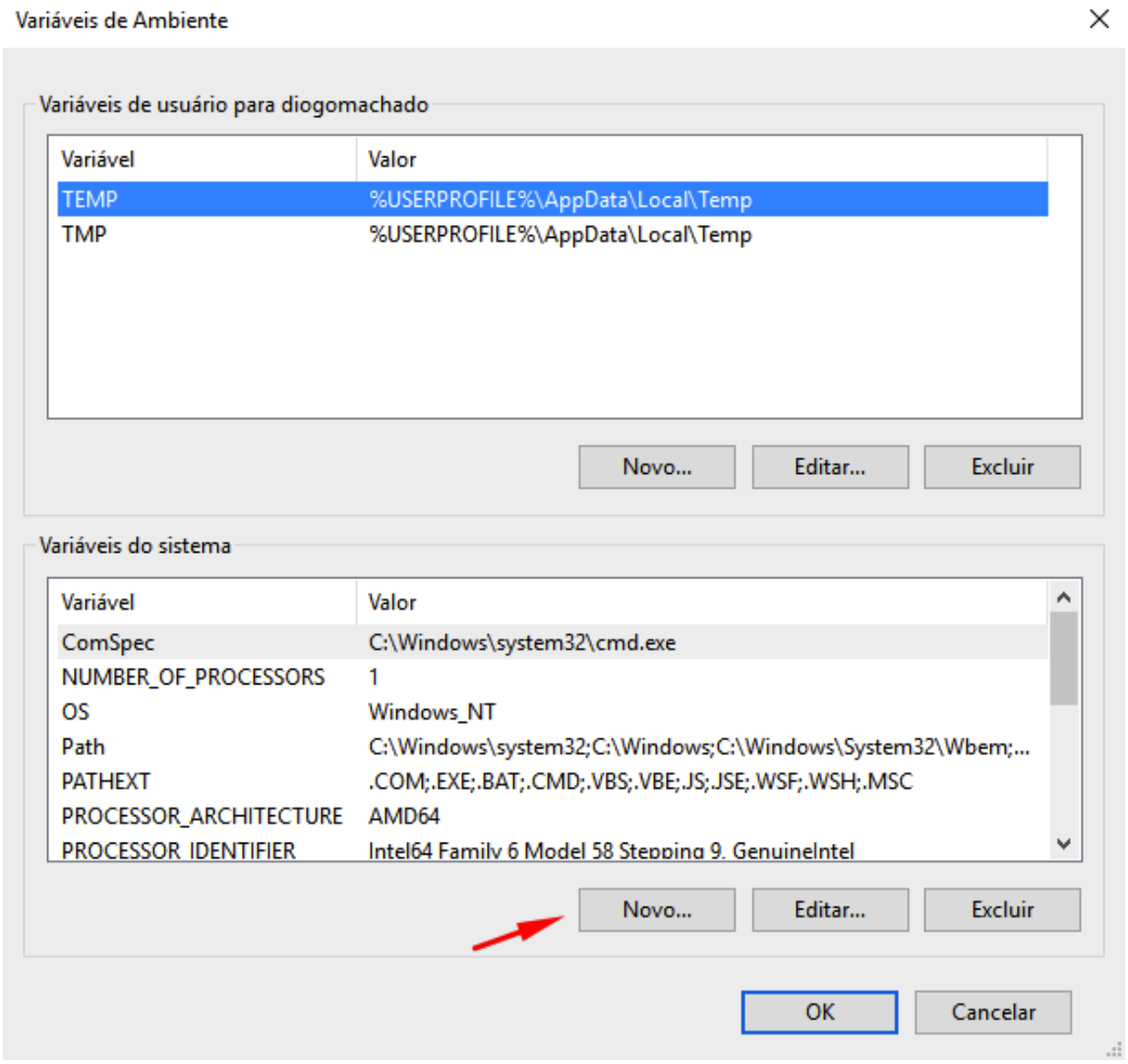

Windows 10 64bits [Executando]
Local (C:) › Usuários › diogomachado › AppData › Local › Android › sdk
Copiar endereço
Copiar endereço como texto
Editar endereço
Excluir histórico
Data de modificaç...
Tipo
Tamanho
06/04/2016 19:16 Pasta de arquivos
06/04/2016 19:18 Pasta de arquivos
06/04/2016 19:18 Pasta de arquivos
06/04/2016 19:16 Pasta de arquivos
06/04/2016 19:18 Pasta de arquivos
06/04/2016 19:17 Pasta de arquivos
06/04/2016 19:18 Pasta de arquivos
06/04/2016 19:16 Pasta de arquivos

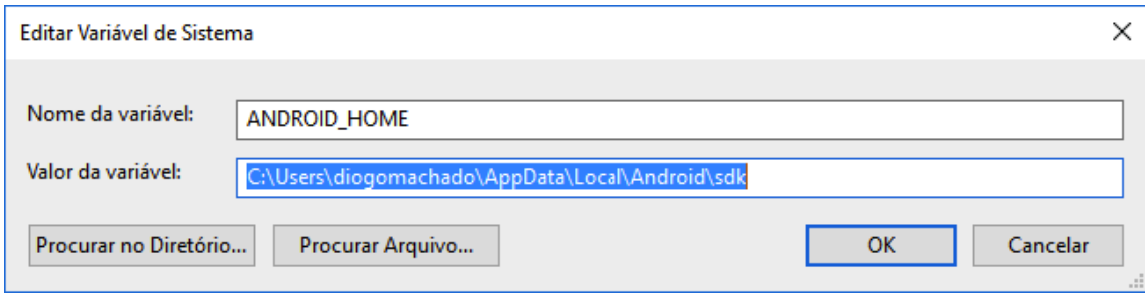
2. Agora, vamos editar as variáveis do sistema.


Melhor correspondência
Editar as variáveis de ambiente do sistema
Painel de controle
Configurações
Editar as variáveis de ambiente para sua conta
Web
variação linguística
variância
variantes linguísticas
variação histórica
variação anatômica
Meus itens
Web
varia

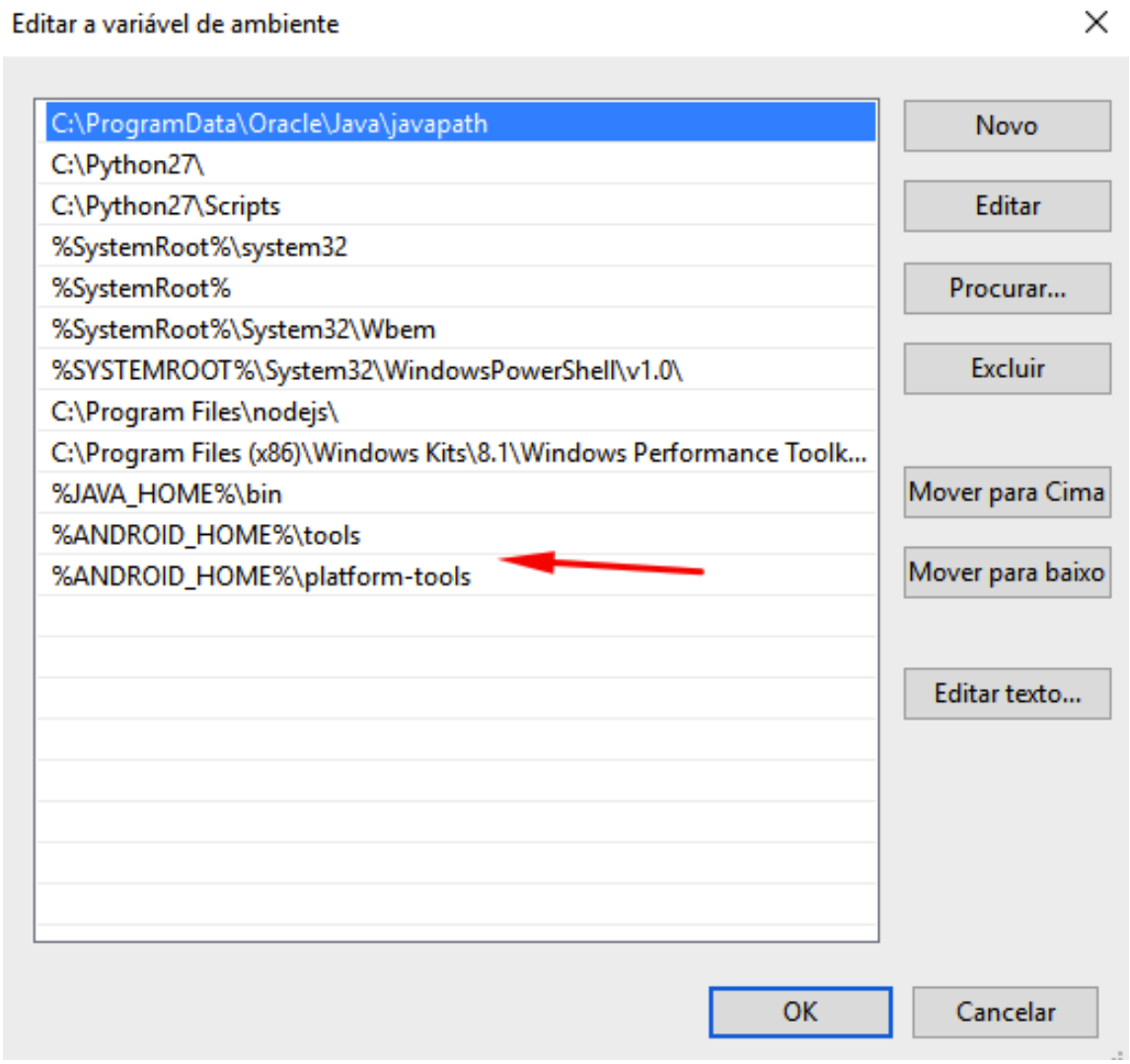


3. Crie uma nova chamada `ANDROID_HOME` .





4. Adicione também ao PATH do sistema.



Feito isso, feche e abra o terminal novamente. Você terá os comandos necessários do Android disponíveis. Você pode testar, executando o seguinte comando: `android` .

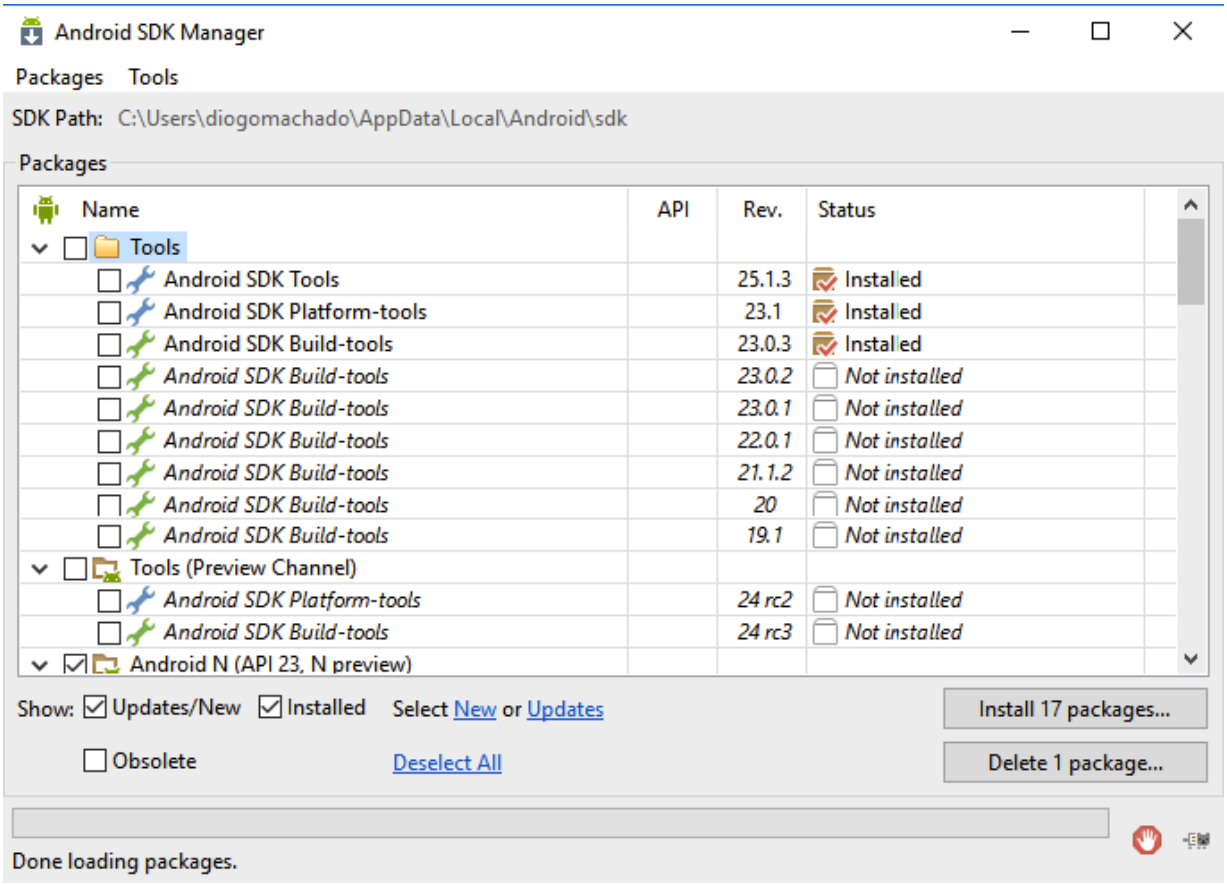


Figura 9.25: Windows Android SDK Manager

Faça o download inicialmente dos pacotes:

- **Android SDK Tools** – Componente com as ferramentas essenciais para emulação no Android.
- **Android SDK Platform-tools** – Inclui várias ferramentas requeridas para a plataforma Android, como o Android Debug Bridge.
- **Android SDK Build-tools** – É um componente necessário que inclui ferramentas para o *build* das aplicações no Android.
- **API Level** – Baixe a última versão da API Level do Android, geralmente descrito no gerenciador com o padrão *nome da versão* mais *número API*, como **Android 6 (API 23)**. É **importantíssimo** que você baixe a última versão mesmo que não queira trabalhar com ela, porque o CLI do Cordova tenta acompanhar sempre a última versão do API Level.

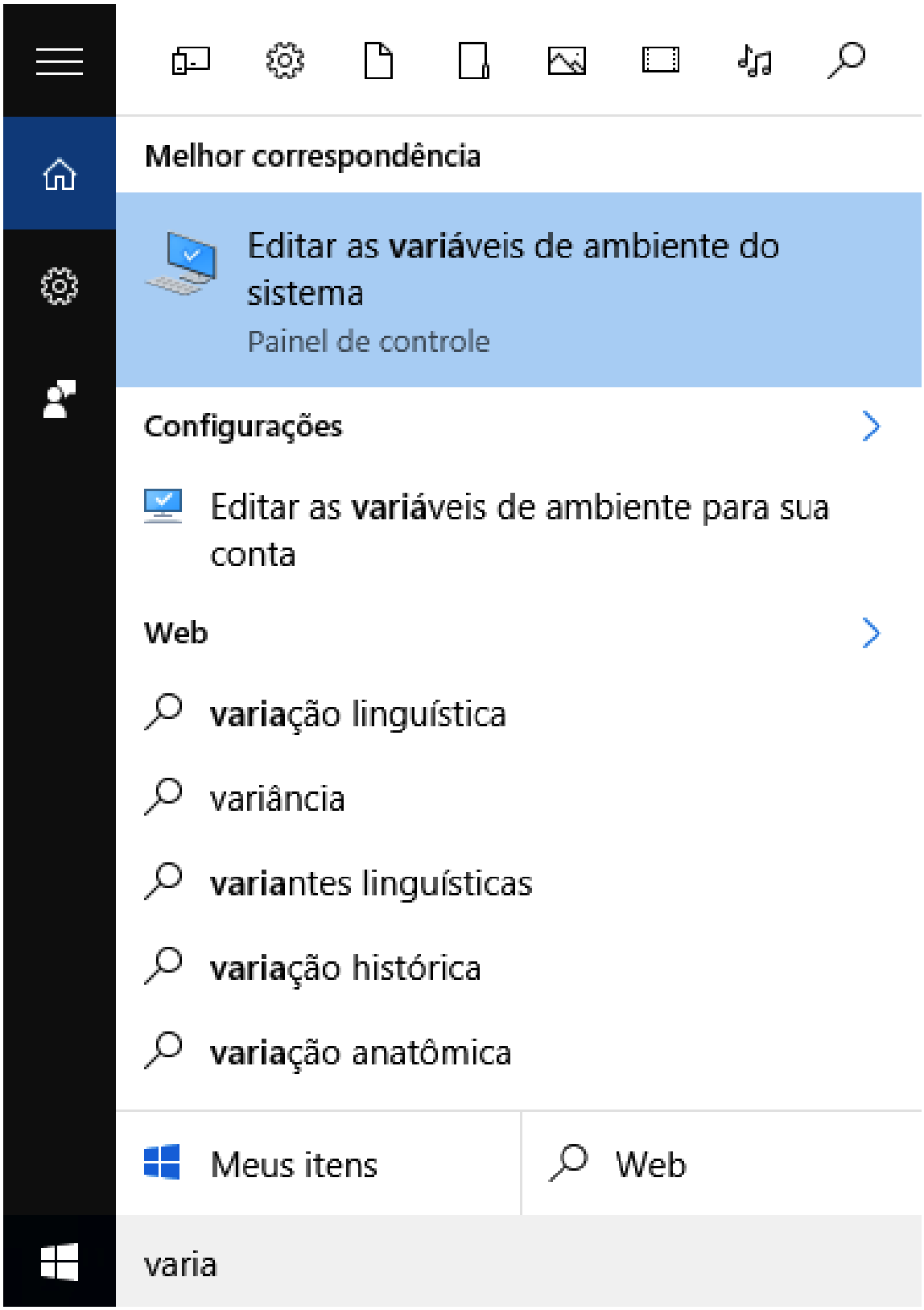
Baixe e instale o Apache Ant

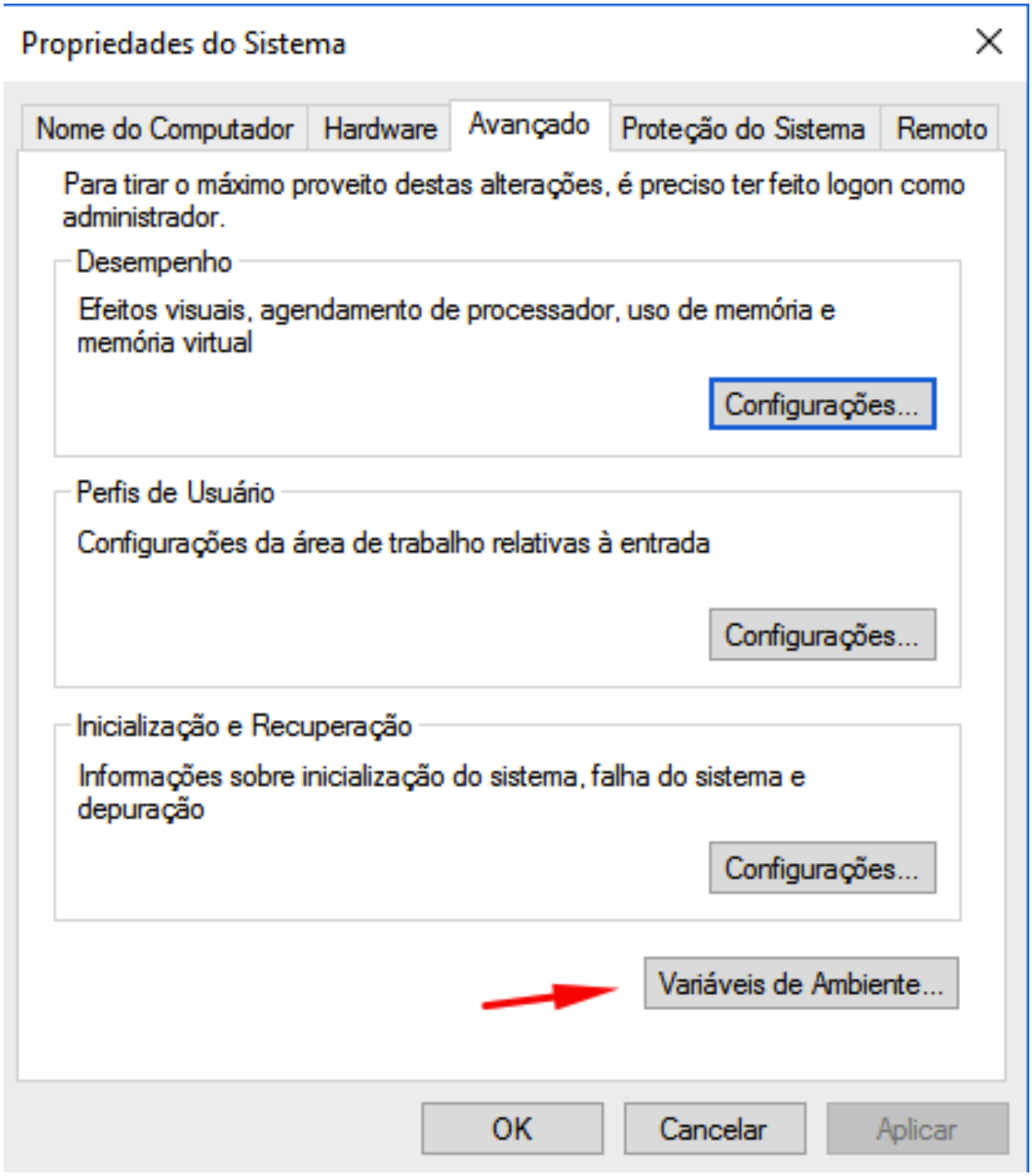
Apache Ant é uma ferramenta utilizada para automação de compilação na construção de software na linguagem Java. É uma biblioteca **indispensável** para o Android SDK fazer a construção (*build*) da aplicação Android. O Ant usa um arquivo XML para descrever o processo de construção – por padrão, chamado `build.xml` .

Acesse http://ant.apache.org, vá à seção `Download -> Binary Distributions` e, em `Current Release of Ant` , você pode baixar o arquivo no formato `.zip` . Feito o download, *unzip* esse arquivo no diretório do seu usuário.

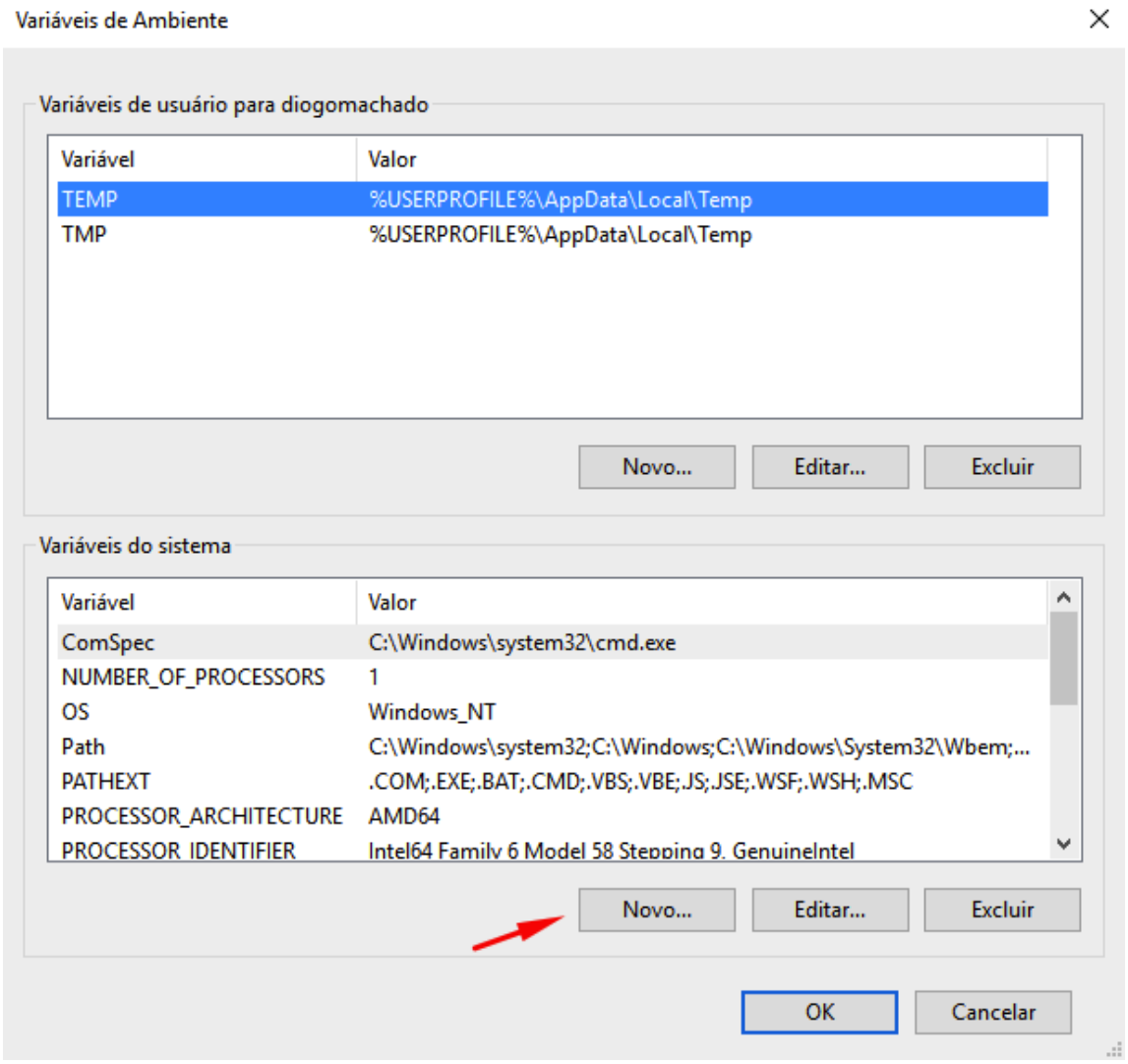
Agora vamos adicionar o Ant ao `PATH` :

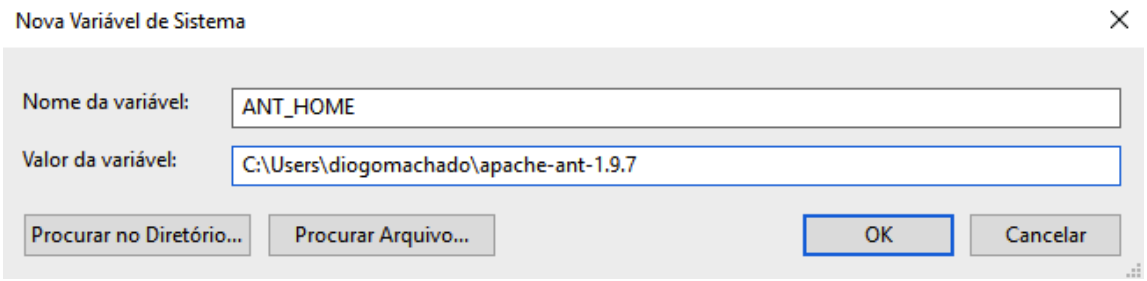
1. Primeiro vamos copiar o endereço do diretório onde está localizado o Ant, no meu caso estava em `C:\Users\diogomachado\apache-ant-1.9.7` .
2. Agora, vamos editar as variáveis do sistema.


Melhor correspondência
Editar as variáveis de ambiente do sistema
Painel de controle
Configurações
Editar as variáveis de ambiente para sua conta
Web
variação linguística
variância
variantes linguísticas
variação histórica
variação anatômica
Meus itens
Web
varia

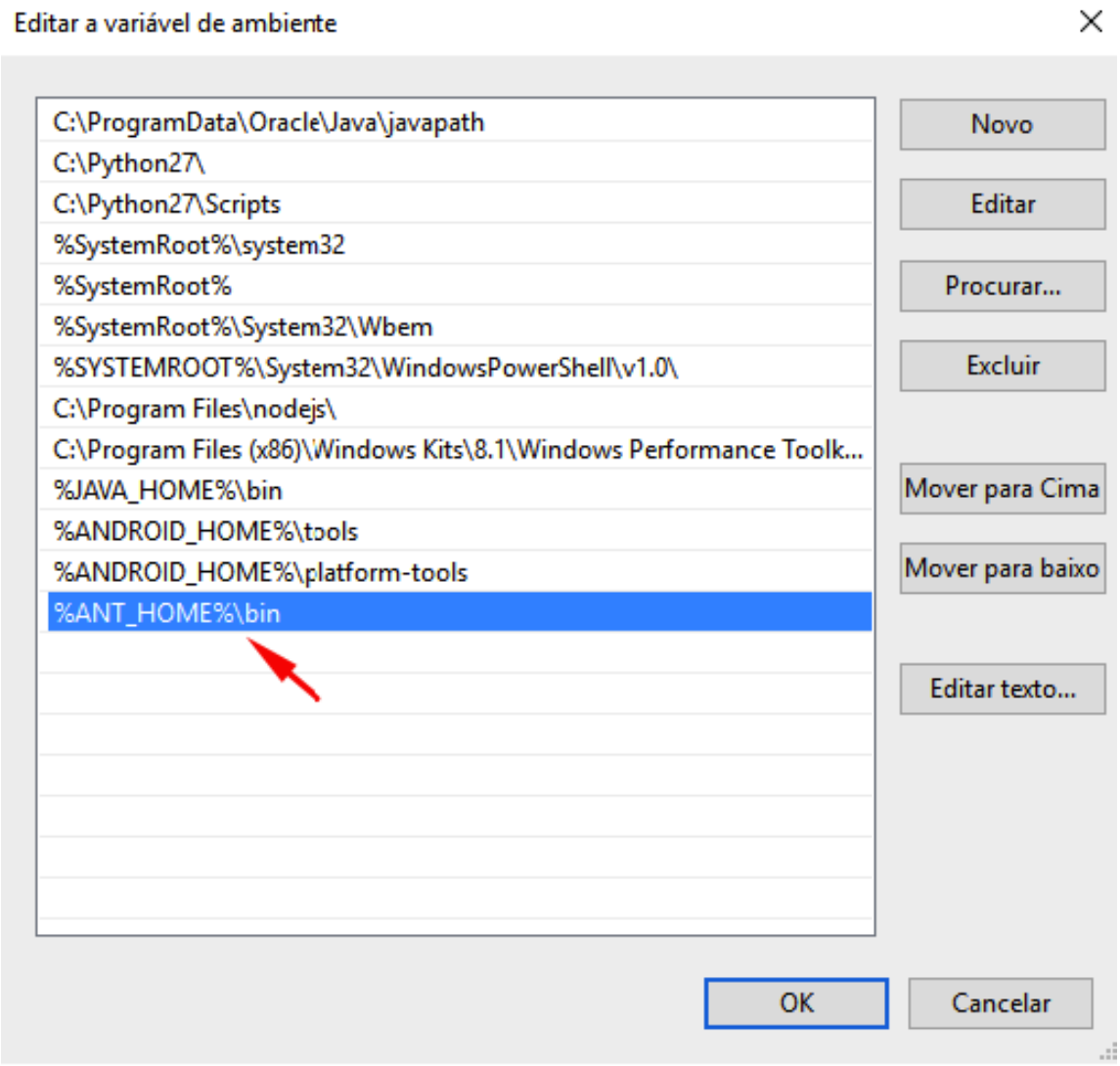


3. Crie uma nova chamada `ANT_HOME` .





4. Adicione também ao PATH do sistema.



Instale o Node.js

Agora vamos instalar o Node.js. Ele é necessário para instalarmos a linha de comando (CLI), tanto do Cordova quanto do PhoneGap. Ele também abre portas para outros pacotes disponibilizados no https://www.npmjs.com que podem ser muito úteis no desenvolvimento híbrido (como o Gulp e o Browser-sync).

Acesse o site https://nodejs.org, baixe o instalador na sua versão LTS e, depois, é só ir avançando.

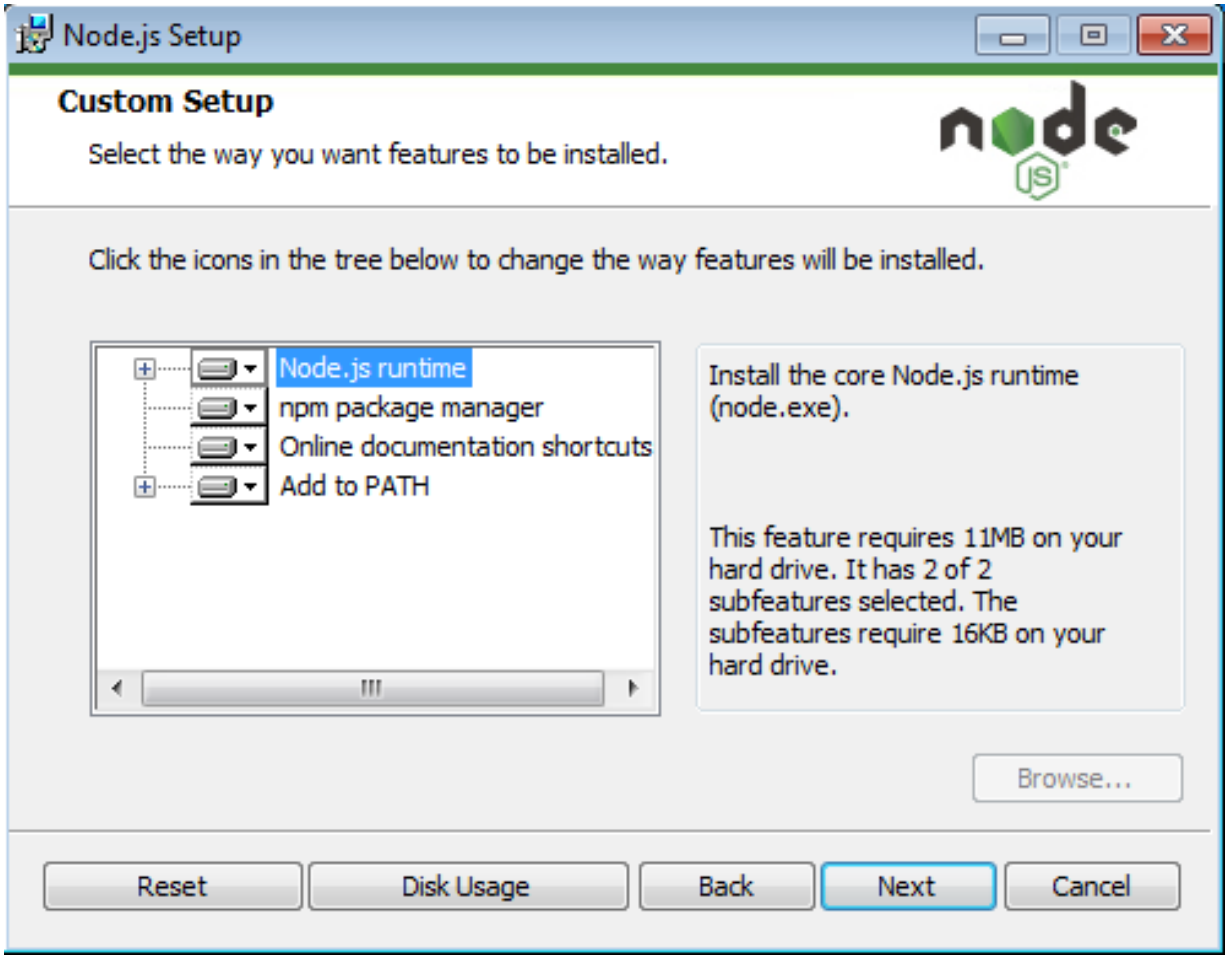

Node.js Setup
Custom Setup
Select the way you want features to be installed.
Click the icons in the tree below to change the way features will be installed.
Node.js runtime
npm package manager
Online documentation shortcuts
Add to PATH
Install the core Node.js runtime (node.exe).
This feature requires 11MB on your hard drive. It has 2 of 2 subfeatures selected. The subfeatures require 16KB on your hard drive.
Browse...
Reset
Disk Usage
Back
Next
Cancel

Figura 9.31: Instalação do Node.js no Windows

Como mostrado na imagem anterior, o instalador já vem com o Node.js e o NPM, e configura o PATH do sistema. Se você pesquisar por Node.js, verá que existe um prompt do Node.js:

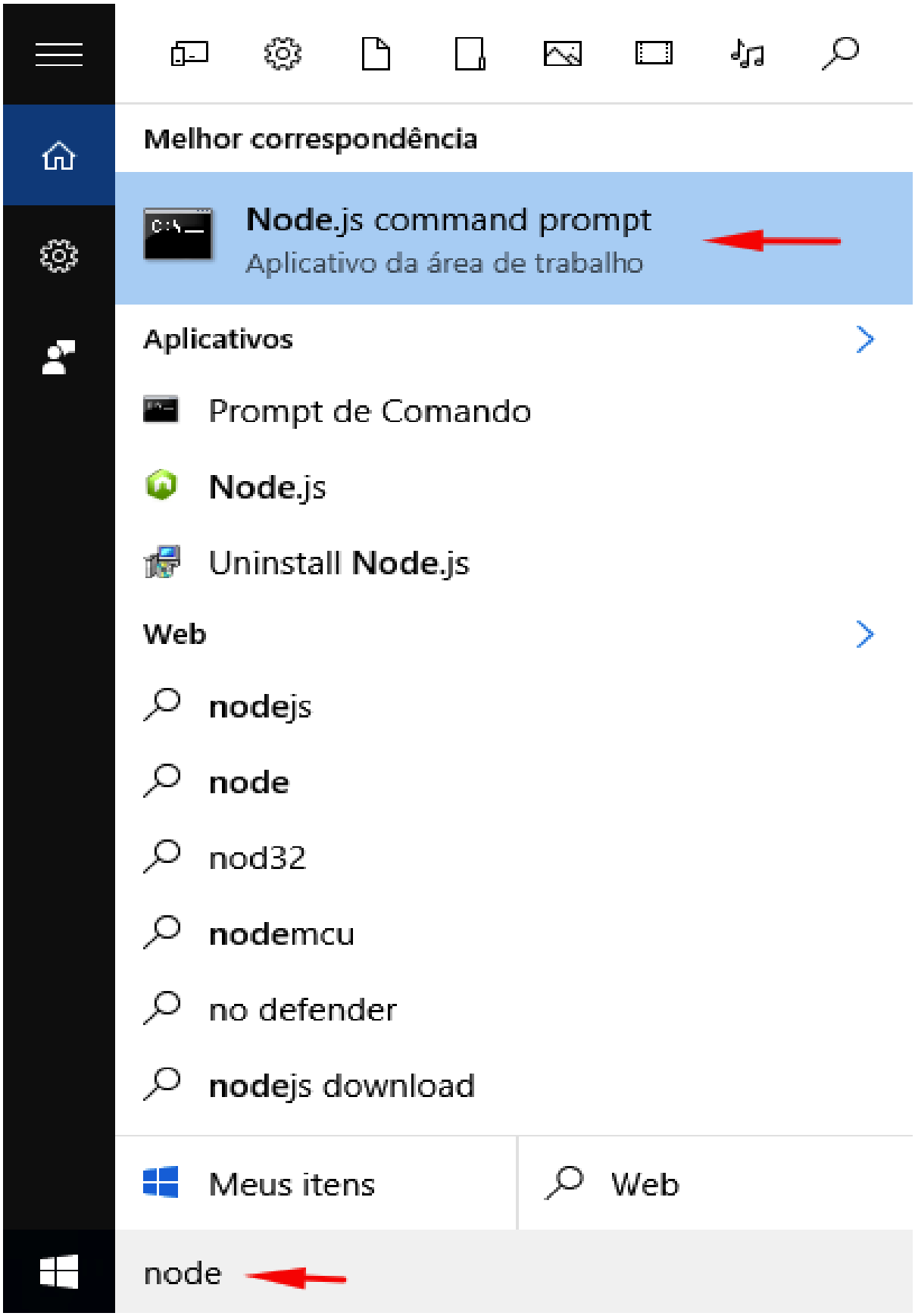

Melhor correspondência
Node.js command prompt
Aplicativo da área de trabalho
Aplicativos
Prompt de Comando
Node.js
Uninstall Node.js
Web
nodejs
node
nod32
nodemcu
no defender
nodejs download
Meus itens
Web
node

Figura 9.32: Prompt do Node.js

Este é um prompt que tenta garantir que o Node.js e o NPM sejão carregados corretamente, mas você pode usar o prompt comum também.

Instale o CLI do Cordova e do PhoneGap

Agora que temos o NPM disponível, podemos instalar o CLI do Cordova e do PhoneGap. No Windows, abra o terminal e execute:

```
sudo npm install -g cordova
```

Diferente dos sistemas Unix, o **Windows não vem com o Python** instalado por padrão. Vamos precisar dele para instalar o CLI do PhoneGap. Portanto, precisamos acessar o site e baixá-lo: https://www.python.org.

1. Vá em `Downloads`.
2. Clique em `Windows`.
3. Neste livro, instalamos a versão Python 2.7.11.

Execute a instalação:

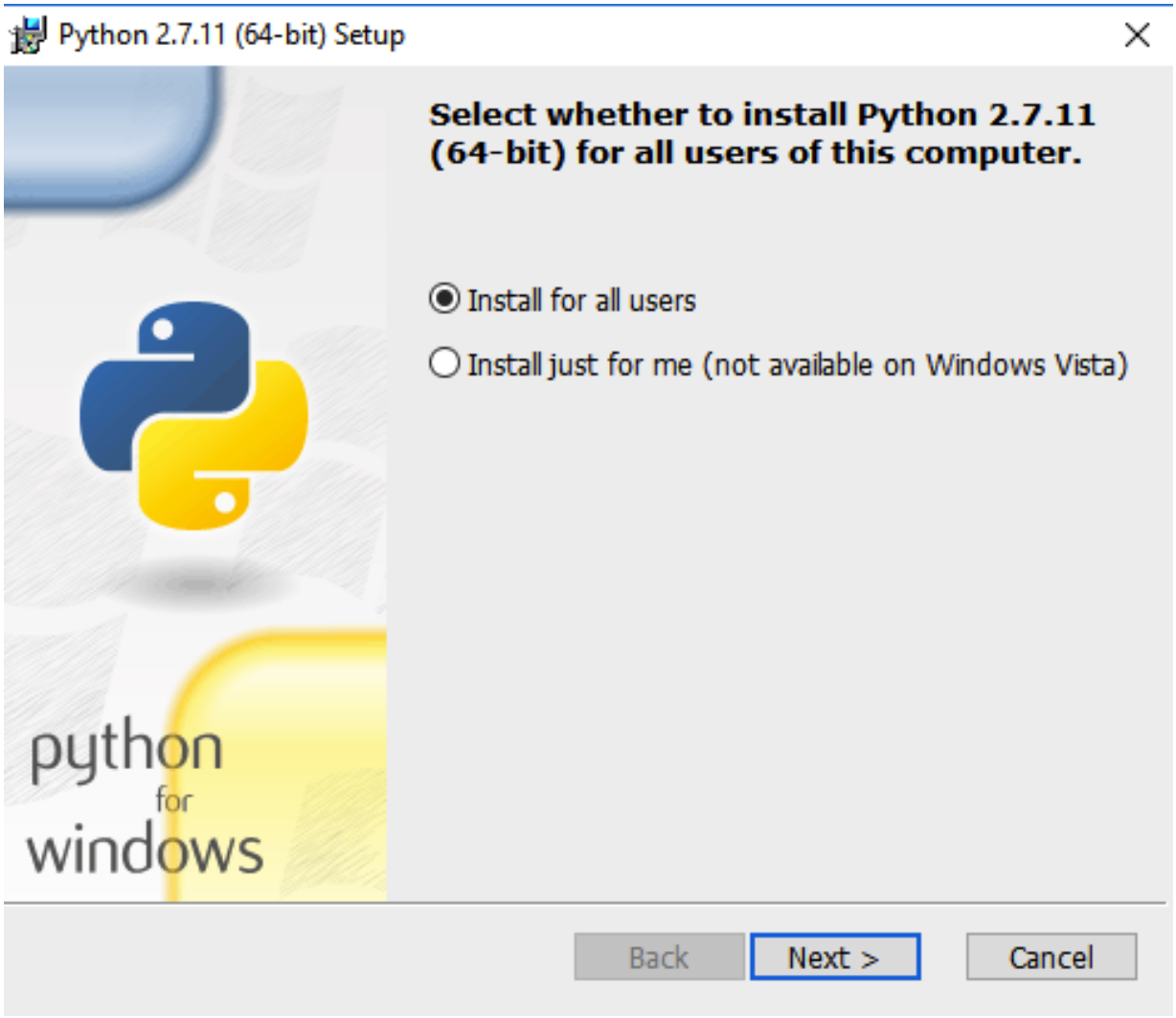


Figura 9.33: Instalação Python

Marque a opção `Add python.exe to Path` :

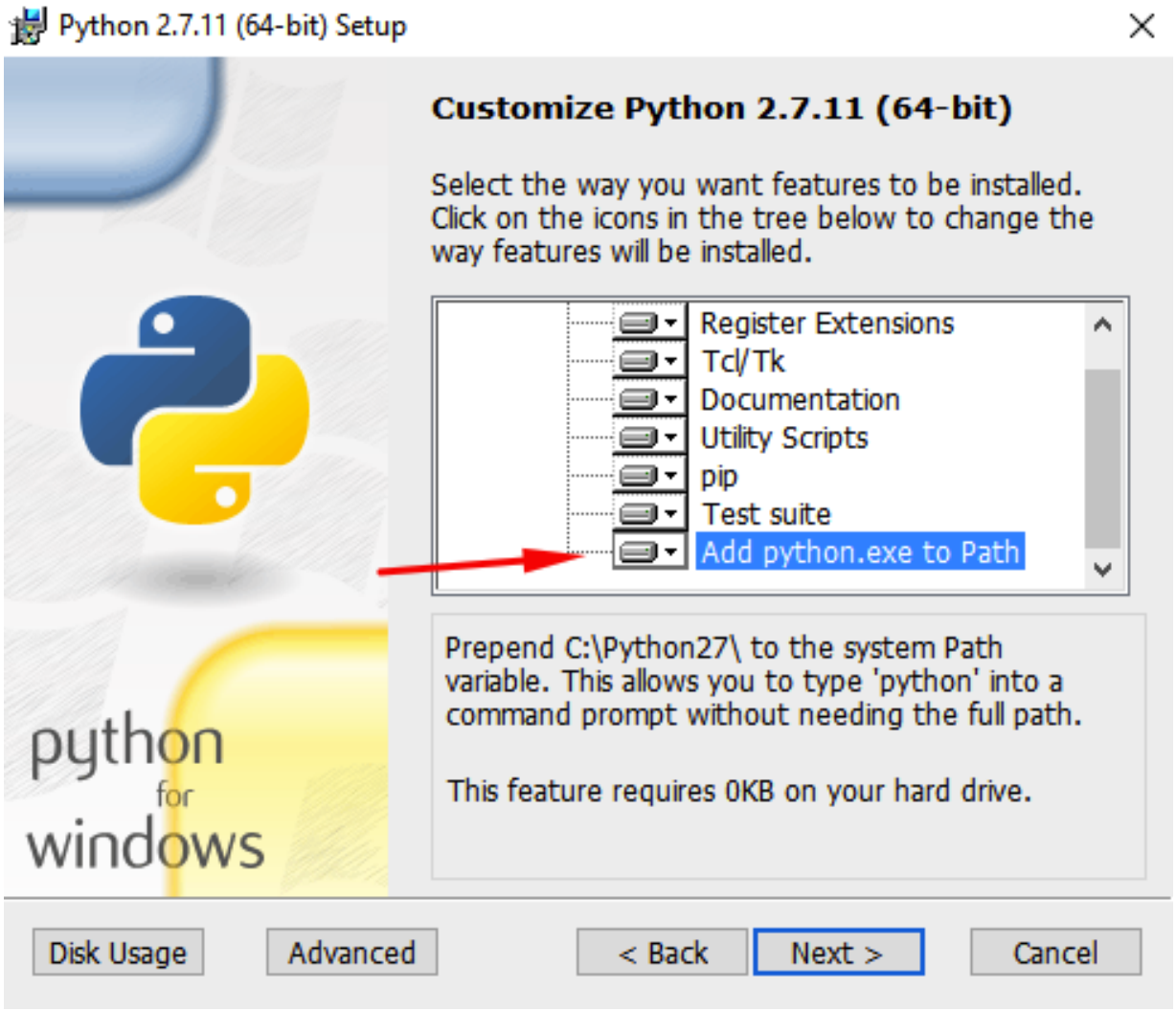


Figura 9.34: Instalação Python - Seleção PATH

Em seguida, instale o PhoneGap também:

```
sudo npm install -g phonegap
```

Relembrando nossos passos:

- Verificamos se o Java SDK já estava instalado; caso contrário, instalamos.
- Definimos a variável `JAVA_HOME` no sistema.
- Adicionamos ao `PATH` a variável `JAVA_HOME` com o diretório `/bin`.
- Baixamos e instalamos o Android Studio e o SDK.

- Configuramos as variáveis do sistema para o `ANDROID_HOME`, e também adicionamos ao `PATH` dois caminhos da SDK.
- Fizemos o download dos pacotes necessários para o futuro `build` da aplicação.
- Baixamos o Apache Ant.
- Configuramos a variável `ANT_HOME`.
- Adicionamos o `ANT_HOME` com o diretório `/bin` ao `PATH`.
- Instalamos o Node.js.
- Por fim, instalamos os pacotes do Cordova e do PhoneGap.

9.4 iOS no macOS

Se você deseja desenvolver para iOS, a primeira coisa que devemos fazer é entrar na *App Store* e baixar o **Xcode**. O Xcode é onde vamos assinar nossa aplicação, fazer configurações avançadas, emular, entre outras funções; é **indispensável** que você o tenha.

Instale o Xcode e baixe os emuladores

Abra a `App Store`, procure por *Xcode* e faça a instalação – até aqui, não tem mistério. O Xcode é um software livre, portanto, de graça.

Ele não vem com os emuladores pré-instalados, por isso, precisamos fazer o download. Basta abrirmos o programa. No canto esquerdo, clique no nome `Xcode -> Preferencias`, vá à aba `Componentes`, e agora é só baixar os emuladores que deseja. Esse processo será demorado, por isso é bom colocar para baixar antes mesmo de começar a trabalhar no seu aplicativo, assim você evita perder tempo depois.

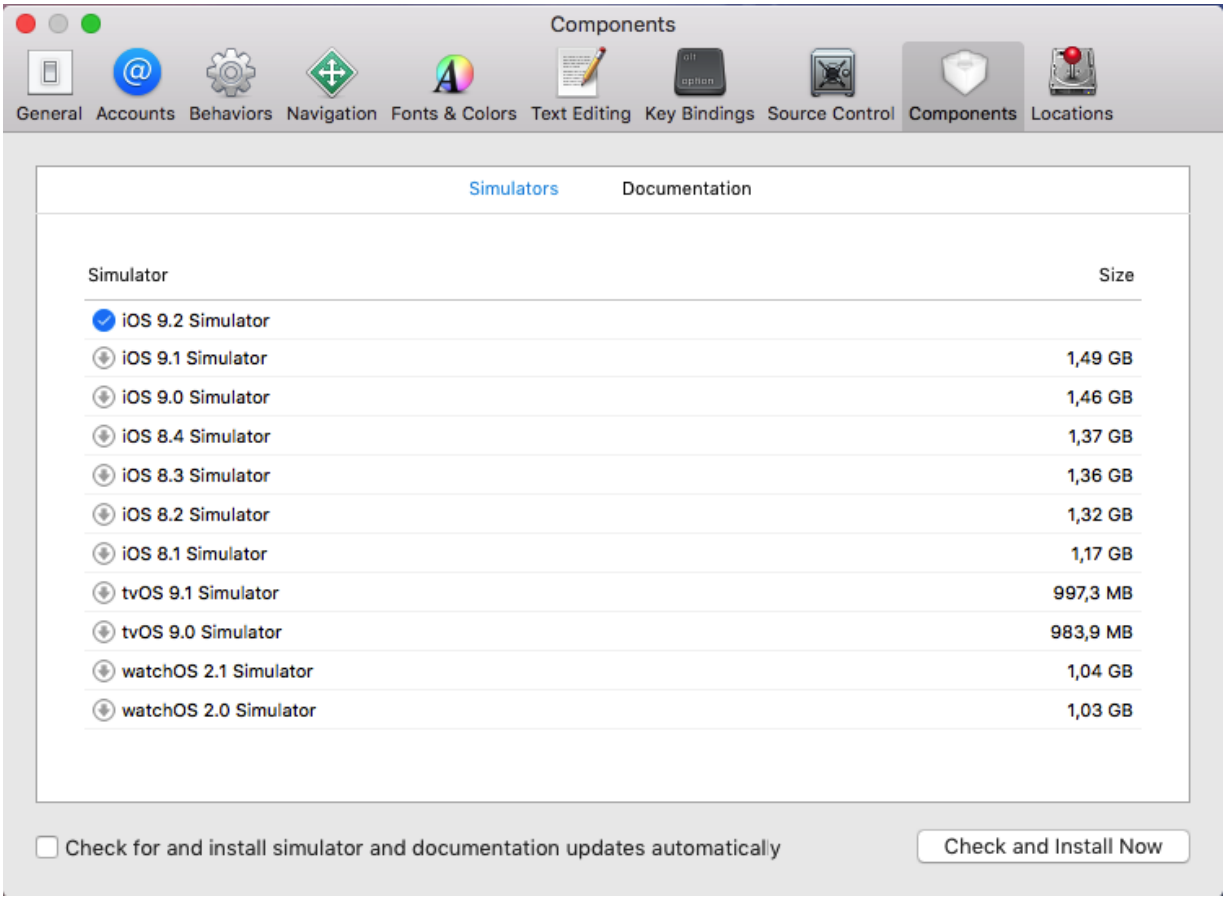


Figura 9.35: Download de emuladores no Xcode

Instalar o Node.js

Acesse o seu site oficial (https://nodejs.org), baixe o instalador da versão *LTS*. Feito isso, dê um duplo clique no instalador e siga as instruções.

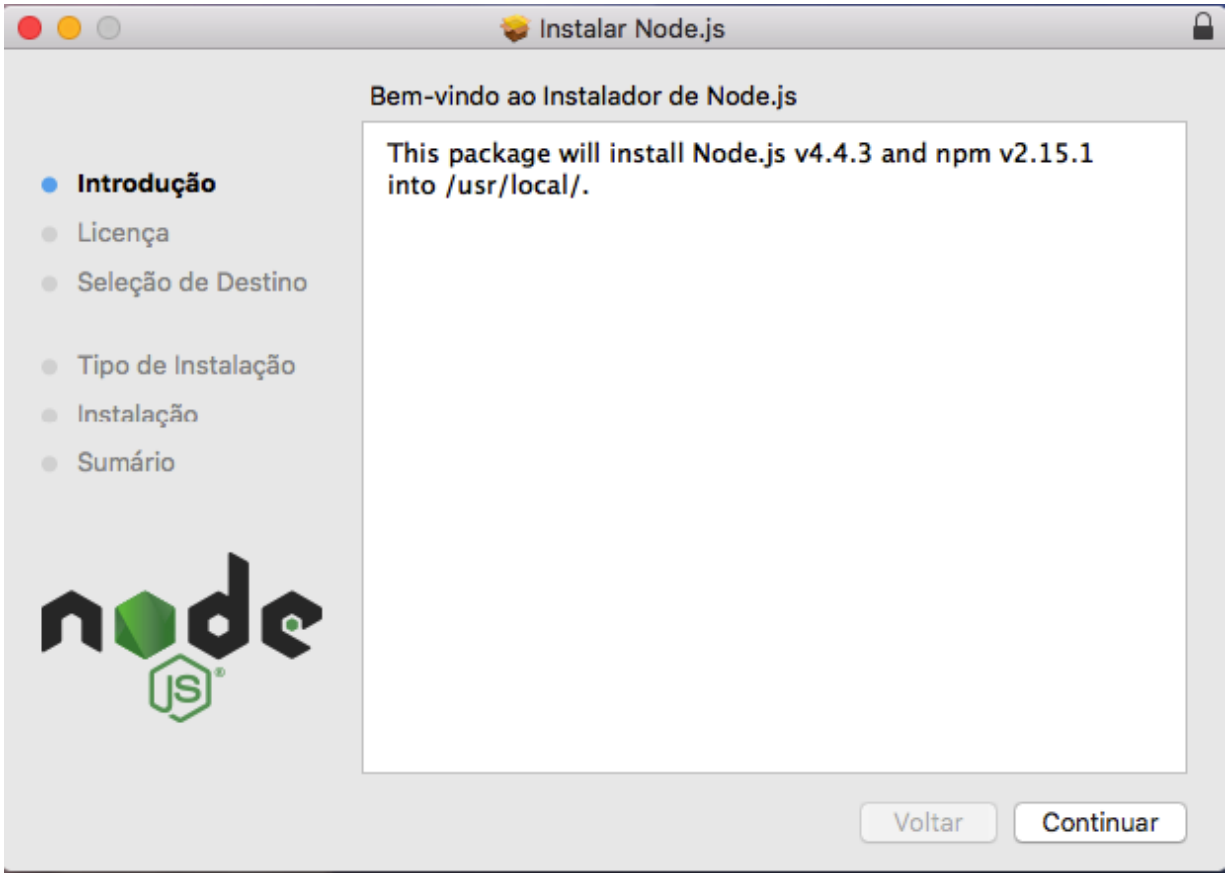


Figura 9.36: Instalação do Node.js

Ele vai instalar o Node.js e o NPM no diretório `/usr/local/`, como também configurar o `PATH`. Isso quer dizer que, logo após a instalação, já teremos os comandos do Node.js e do NPM disponíveis no terminal.

Instale o CLI do Cordova e do PhoneGap

Agora que temos o NPM disponível, podemos instalar o CLI do Cordova e do PhoneGap:

```
sudo npm install -g cordova
```

Em seguida, instalaremos o PhoneGap também:

```
sudo npm install -g phonegap
```

Feito isso, sua máquina está configurada para emular e fazer `build` na plataforma iOS.

9.5 iOS no Windows

Não tem jeito, você precisa comprar um Macbook. Os motivos são:

1. Você precisará do Xcode, disponível apenas no sistema operacional macOS da Apple;
2. Para emular seu app no iOS, o Xcode também será necessário;
3. Para fazer a assinatura do app, processo que precede a subida do app para a loja, você vai precisar do Xcode;
4. Adivinhe o que é necessário para fazermos configurações avançadas? Pois é, o Xcode.

Ou seja, precisamos dele e ele só roda no macOS no Mac.

Todos desenvolvedores sabemos que aqui, no Brasil, devido à alta taxa de impostos (e outros motivos), os produtos da Apple estão cada vez mais caros. Mas se você deseja desenvolver para a plataforma, você pode:

1. Comprar um Macbook usado;
2. Alugar uma máquina virtual com macOS na internet, que pode ser uma solução rápida. Porém, no longo prazo, é melhor comprar um Mac.

Portanto, se você deseja desenvolver para essa plataforma, já solicite um Macbook ao seu chefe, ou se planeje para adquirir um.

9.6 iOS no Linux

Não tem jeito, você precisa comprar um Macbook. É só ler as mesmas explicações da seção anterior.

9.7 Revisão

Passamos por muitas configurações. Aqui, a intenção era lhe passar tudo o que precisa ser feito para deixar seu ambiente pronto para testes.